encrypt all transports

@feyeleanor

https

```
package main

import . "fmt"
import . "net/http"

const ADDRESS = ":443"

func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })
  ListenAndServeTLS(ADDRESS, "cert.pem", "key.pem", nil)
}
```

```
package main

import . "fmt"
import . "net/http"

const ADDRESS = ":443"

func main() {
  message := "hello world"
  HandleFunc("/hello", func(w ResponseWriter, r *Request) {
    w.Header().Set("Content-Type", "text/plain")
    Fprintf(w, message)
  })
  ListenAndServeTLS(ADDRESS, "cert.pem", "key.pem", nil)
}
```

tcp/tls server

```
package main

import "crypto/rand"
import "crypto/tls"
import . "fmt"


func main() {
  Listen(":443", ConfigTLS("scert", "skey"), func(c *tls.Conn) {
    Fprintln(c, "hello world")
  })
}


func Listen(a string, conf *tls.Config, f func(*tls.Conn)) {
  if listener, e := tls.Listen("tcp", a, conf); e == nil {
    for {
      if connection, e := listener.Accept(); e == nil {
        go func(c *tls.Conn) {
          defer c.Close()
          f(c)
        }(connection.(*tls.Conn))
      }
    }
  }
}
```

```
func ConfigTLS(c, k string) (r *tls.Config) {
  if cert, e := tls.LoadX509KeyPair(c, k); e == nil {
    r = &tls.Config{
      Certificates: []tls.Certificate{ cert },
      Rand: rand.Reader,
    }
  }
  return
}
```

```
package main

import "crypto/rand"
import "crypto/tls"
import . "fmt"


func main() {
  Listen(":443", ConfigTLS("scert", "skey"), func(c *tls.Conn) {
    Fprintln(c, "hello world")
  })
}


func Listen(a string, conf *tls.Config, f func(*tls.Conn)) {
  if listener, e := tls.Listen("tcp", a, conf); e == nil {
    for {
      if connection, e := listener.Accept(); e == nil {
        go func(c *tls.Conn) {
          defer c.Close()
          f(c)
        }(connection.(*tls.Conn))
      }
    }
  }
}
```

```
func ConfigTLS(c, k string) (r *tls.Config) {
  if cert, e := tls.LoadX509KeyPair(c, k); e == nil {
    r = &tls.Config{
      Certificates: []tls.Certificate{ cert },
      Rand: rand.Reader,
    }
  }
  return
}
```

tcp/tls client

```
package main

import . "fmt"
import "bufio"
import "net"
import “crypto/tls"

func main() {
  Dial(":1025", ConfigTLS("ccert", "ckey"), func(c net.Conn) {
    if m, e := bufio.NewReader(c).ReadString('\n'); e == nil {
      Printf(m)
    }
  })
}


func ConfigTLS(c, k string) (r *tls.Config) {
  if cert, e := tls.LoadX509KeyPair(c, k); e == nil {
    r = &tls.Config{
      Certificates: []tls.Certificate{ cert },
      InsecureSkipVerify: false,
    }
  }
  return
}
```

```
func Dial(a string, conf *tls.Config, f func(net.Conn)) {
  if c, e := tls.Dial("tcp", a, conf); e == nil {
    defer c.Close()
    f(c)
  }
}
```

```
package main

import . "fmt"
import "bufio"
import "net"
import “crypto/tls"

func main() {
  Dial(":1025", ConfigTLS("ccert", "ckey"), func(c net.Conn) {
    if m, e := bufio.NewReader(c).ReadString('\n'); e == nil {
      Printf(m)
    }
  })
}


func ConfigTLS(c, k string) (r *tls.Config) {
  if cert, e := tls.LoadX509KeyPair(c, k); e == nil {
    r = &tls.Config{
      Certificates: []tls.Certificate{ cert },
      InsecureSkipVerify: true,
    }
  }
  return
}
```

```
func Dial(a string, conf *tls.Config, f func(net.Conn)) {
  if c, e := tls.Dial("tcp", a, conf); e == nil {
    defer c.Close()
    f(c)
  }
}
```

udp/aes server

```
package main

import "crypto/aes"
import "crypto/cipher"
import "crypto/rand"
import . "net"

const AES_KEY = "0123456789012345"

func main() {
  Serve(":1025", func(c *UDPConn, a *UDPAddr, b []byte) {
    if m, e := Encrypt("Hello World", AES_KEY); e == nil {
      c.WriteToUDP(m, a)
    }
  })
}

func Serve(a string, f func(*UDPConn, *UDPAddr, []byte)) {
  if address, e := ResolveUDPAddr("udp", a); e == nil {
    if conn, e := ListenUDP("udp", address); e == nil {
      for b := make([]byte, 1024); ; b = make([]byte, 1024) {
        if n, client, e := conn.ReadFromUDP(b); e == nil {
          go f(conn, client, b[:n])
        }
      }
    }
  }
  return
}
```

```
func Quantise(m string) (b []byte, e error) {
  b = append(b, m...)
  if p := len(b) % aes.BlockSize; p != 0 {
    p = aes.BlockSize - p
    //  this is insecure and inflexible as we're padding with NUL
   b = append(b, make([]byte, p)...)
  }
  return
}

func IV() (b []byte, e error) {
  b = make([]byte, aes.BlockSize)
  _, e = rand.Read(b)
  return
}

func Encrypt(m, k string) (o []byte, e error) {
  if o, e = Quantise([]byte(m)); e == nil {
    var b cipher.Block
    if b, e = aes.NewCipher([]byte(k)); e == nil {
      var iv []byte
      if iv, e = IV(); e == nil {
        c := cipher.NewCBCEncrypter(b, iv)
        c.CryptBlocks(o, o)
        o = append(iv, o...)
      }
    }
  }
  return
}
```

```
package main

import "crypto/aes"
import "crypto/cipher"
import "crypto/rand"
import . "net"

const AES_KEY = "0123456789012345"

func main() {
  Serve(":1025", func(c *UDPConn, a *UDPAddr, b []byte) {
    if m, e := Encrypt("Hello World", AES_KEY); e == nil {
      c.WriteToUDP(m, a)
    }
  })
}

func Serve(a string, f func(*UDPConn, *UDPAddr, []byte)) {
  if address, e := ResolveUDPAddr("udp", a); e == nil {
    if conn, e := ListenUDP("udp", address); e == nil {
      for b := make([]byte, 1024); ; b = make([]byte, 1024) {
        if n, client, e := conn.ReadFromUDP(b); e == nil {
          go f(conn, client, b[:n])
        }
      }
    }
  }
  return
}
```

```
func Quantise(m string) (b []byte, e error) {
  b = append(b, m...)
  if p := len(b) % aes.BlockSize; p != 0 {
    p = aes.BlockSize - p
    //  this is insecure and inflexible as we're padding with NUL
   b = append(b, make([]byte, p)...)
  }
  return
}

func IV() (b []byte, e error) {
  b = make([]byte, aes.BlockSize)
  _, e = rand.Read(b)
  return
}

func Encrypt(m, k string) (o []byte, e error) {
  if o, e = Quantise([]byte(m)); e == nil {
    var b cipher.Block
    if b, e = aes.NewCipher([]byte(k)); e == nil {
      var iv []byte
      if iv, e = IV(); e == nil {
        c := cipher.NewCBCEncrypter(b, iv)
        c.CryptBlocks(o, o)
        o = append(iv, o...)
      }
    }
  }
  return
}
```

```
package main

import "crypto/aes"
import "crypto/cipher"
import "crypto/rand"
import . "net"

const AES_KEY = "0123456789012345"

func main() {
  Serve(":1025", func(c *UDPConn, a *UDPAddr, b []byte) {
    if m, e := Encrypt("Hello World", AES_KEY); e == nil {
      c.WriteToUDP(m, a)
    }
  })
}

func Serve(a string, f func(*UDPConn, *UDPAddr, []byte)) {
  if address, e := ResolveUDPAddr("udp", a); e == nil {
    if conn, e := ListenUDP("udp", address); e == nil {
      for b := make([]byte, 1024); ; b = make([]byte, 1024) {
        if n, client, e := conn.ReadFromUDP(b); e == nil {
          go f(conn, client, b[:n])
        }
      }
    }
  }
  return
}
```

```
func Quantise(m string) (b []byte, e error) {
  b = append(b, m...)
  if p := len(b) % aes.BlockSize; p != 0 {
    p = aes.BlockSize - p
    //  this is insecure and inflexible as we're padding with NUL
   b = append(b, make([]byte, p)...)
  }
  return
}

func IV() (b []byte, e error) {
  b = make([]byte, aes.BlockSize)
  _, e = rand.Read(b)
  return
}

func Encrypt(m, k string) (o []byte, e error) {
  if o, e = Quantise([]byte(m)); e == nil {
    var b cipher.Block
    if b, e = aes.NewCipher([]byte(k)); e == nil {
      var iv []byte
      if iv, e = IV(); e == nil {
        c := cipher.NewCBCEncrypter(b, iv)
        c.CryptBlocks(o, o)
        o = append(iv, o...)
      }
    }
  }
  return
}
```

```
package main

import "crypto/aes"
import "crypto/cipher"
import "crypto/rand"
import . "net"

const AES_KEY = "0123456789012345"

func main() {
  Serve(":1025", func(c *UDPConn, a *UDPAddr, b []byte) {
    if m, e := Encrypt("Hello World", AES_KEY); e == nil {
      c.WriteToUDP(m, a)
    }
  })
}

func Serve(a string, f func(*UDPConn, *UDPAddr, []byte)) {
  if address, e := ResolveUDPAddr("udp", a); e == nil {
    if conn, e := ListenUDP("udp", address); e == nil {
      for b := make([]byte, 1024); ; b = make([]byte, 1024) {
        if n, client, e := conn.ReadFromUDP(b); e == nil {
          go f(conn, client, b[:n])
        }
      }
    }
  }
  return
}
```

```
func Quantise(m string) (b []byte, e error) {
  b = append(b, m...)
  if p := len(b) % aes.BlockSize; p != 0 {
    p = aes.BlockSize - p
    //  this is insecure and inflexible as we're padding with NUL
   b = append(b, make([]byte, p)...)
  }
  return
}

func IV() (b []byte, e error) {
  b = make([]byte, aes.BlockSize)
  _, e = rand.Read(b)
  return
}

func Encrypt(m, k string) (o []byte, e error) {
  if o, e = Quantise([]byte(m)); e == nil {
    var b cipher.Block
    if b, e = aes.NewCipher([]byte(k)); e == nil {
      var iv []byte
      if iv, e = IV(); e == nil {
        c := cipher.NewCBCEncrypter(b, iv)
        c.CryptBlocks(o, o)
        o = append(iv, o...)
      }
    }
  }
  return
}
```

udp/aes client

```
package main

import "bufio"
import "crypto/cipher"
import "crypto/aes"
import . "fmt"
import . "net"

const AES_KEY = "0123456789012345"


func main() {
  Request(":1025", func(c *UDPConn) {
    c.Write(make([]byte, 1))
    if m, e := ReadStream(c); e == nil {
      if m, e := Decrypt(m, AES_KEY); e == nil {
        Println(string(m))
      }
    }
  })
}


func Decrypt(m []byte, k string) (r string, e error) {
  var b cipher.Block
  if b, e = aes.NewCipher([]byte(k)); e == nil {
    var iv []byte
    iv, m = Unpack(m)
    c := cipher.NewCBCDecrypter(b, iv)
    c.CryptBlocks(m, m)
    r = Dequantise(m)
  }
  return
}
```

```
func Unpack(m []byte) (iv, r []byte) {
  return m[:aes.BlockSize], m[aes.BlockSize:]
}


func Dequantise(m []byte) string {
  var i int
  for i = len(m) - 1; i > 0 && m[i] == 0; i-- {}
  return string(m[:i + 1])
}


func Request(a string, f func(Conn)) {
  if address, e := ResolveUDPAddr("udp", a); e == nil {
    if conn, e := DialUDP("udp", nil, address); e == nil {
      defer conn.Close()
      f(conn)
    }
  }
}
```

```
package main

import "bufio"
import "crypto/cipher"
import "crypto/aes"
import . "fmt"
import . "net"

const AES_KEY = "0123456789012345"


func main() {
  Request(":1025", func(c *UDPConn) {
    c.Write(make([]byte, 1))
    if m, e := ReadStream(c); e == nil {
      if m, e := Decrypt(m, AES_KEY); e == nil {
        Println(string(m))
      }
    }
  })
}


func Decrypt(m []byte, k string) (r string, e error) {
  var b cipher.Block
  if b, e = aes.NewCipher([]byte(k)); e == nil {
    var iv []byte
    iv, m = Unpack(m)
    c := cipher.NewCBCDecrypter(b, iv)
    c.CryptBlocks(m, m)
    r = Dequantise(m)
  }
  return
}
```

```
func Unpack(m []byte) (iv, r []byte) {
  return m[:aes.BlockSize], m[aes.BlockSize:]
}


func Dequantise(m []byte) string {
  var i int
  for i = len(m) - 1; i > 0 && m[i] == 0; i-- {}
  return string(m[:i + 1])
}


func Request(a string, f func(Conn)) {
  if address, e := ResolveUDPAddr("udp", a); e == nil {
    if conn, e := DialUDP("udp", nil, address); e == nil {
      defer conn.Close()
      f(conn)
    }
  }
}
```

```
package main

import "bufio"
import "crypto/cipher"
import "crypto/aes"
import . "fmt"
import . "net"

const AES_KEY = "0123456789012345"


func main() {
  Request(":1025", func(c *UDPConn) {
    c.Write(make([]byte, 1))
    if m, e := ReadStream(c); e == nil {
      if m, e := Decrypt(m, AES_KEY); e == nil {
        Println(string(m))
      }
    }
  })
}


func Decrypt(m []byte, k string) (r string, e error) {
  var b cipher.Block
  if b, e = aes.NewCipher([]byte(k)); e == nil {
    var iv []byte
    iv, m = Unpack(m)
    c := cipher.NewCBCDecrypter(b, iv)
    c.CryptBlocks(m, m)
    r = Dequantise(m)
  }
  return
}
```

```
func Unpack(m []byte) (iv, r []byte) {
  return m[:aes.BlockSize], m[aes.BlockSize:]
}


func Dequantise(m []byte) string {
  var i int
  for i = len(m) - 1; i > 0 && m[i] == 0; i-- {}
  return string(m[:i + 1])
}


func Request(a string, f func(Conn)) {
  if address, e := ResolveUDPAddr("udp", a); e == nil {
    if conn, e := DialUDP("udp", nil, address); e == nil {
      defer conn.Close()
      f(conn)
    }
  }
}
```

```
package main

import "bufio"
import "crypto/cipher"
import "crypto/aes"
import . "fmt"
import . "net"

const AES_KEY = "0123456789012345"


func main() {
  Request(":1025", func(c *UDPConn) {
    c.Write(make([]byte, 1))
    if m, e := ReadStream(c); e == nil {
      if m, e := Decrypt(m, AES_KEY); e == nil {
        Println(string(m))
      }
    }
  })
}


func Decrypt(m []byte, k string) (r string, e error) {
  var b cipher.Block
  if b, e = aes.NewCipher([]byte(k)); e == nil {
    var iv []byte
    iv, m = Unpack(m)
    c := cipher.NewCBCDecrypter(b, iv)
    c.CryptBlocks(m, m)
    r = Dequantise(m)
  }
  return
}
```

```
func Unpack(m []byte) (iv, r []byte) {
  return m[:aes.BlockSize], m[aes.BlockSize:]
}


func Dequantise(m []byte) string {
  var i int
  for i = len(m) - 1; i > 0 && m[i] == 0; i-- {}
  return string(m[:i + 1])
}


func Request(a string, f func(Conn)) {
  if address, e := ResolveUDPAddr("udp", a); e == nil {
    if conn, e := DialUDP("udp", nil, address); e == nil {
      defer conn.Close()
      f(conn)
    }
  }
}
```

udp/rsa server

```
package main

import . "bytes"
import "crypto/rsa"
import "encoding/gob"
import "net"


func main() {
  HELLO_WORLD := []byte("Hello World")
  RSA_LABEL := []byte("served")
  Serve(":1025", func(c *net.UDPConn, a *net.UDPAddr, b []byte) {
    var key rsa.PublicKey
    if e := gob.NewDecoder(NewBuffer(b)).Decode(&key); e == nil {
      if m, e := Encrypt(&key, HELLO_WORLD, RSA_LABEL); e == nil {
        c.WriteToUDP(m, a)
      }
    }
    return
  })
}


func Encrypt(key *rsa.PublicKey, m, l []byte) ([]byte, error) {
  return rsa.EncryptOAEP(sha1.New(), rand.Reader, key, m, l)
}
```

```
func Serve(a string, f func(*UDPConn, *UDPAddr, []byte)) {
  if address, e := ResolveUDPAddr("udp", a); e == nil {
    if conn, e := ListenUDP("udp", address); e == nil {
      for b := make([]byte, 1024); ; b = make([]byte, 1024) {
        if n, client, e := conn.ReadFromUDP(b); e == nil {
          go f(conn, client, b[:n])
        }
      }
    }
  }
  return
}
```

udp/rsa client

```
package main

import "crypto/rsa"
import "crypto/rand"
import "crypto/sha1"
import "crypto/x509"
import "bytes"
import "encoding/gob"
import "encoding/pem"
import “io/ioutil"
import . "fmt"
import . "net"


func main() {
  Request(":1025", "ckey", func(c *net.UDPConn, k *rsa.PrivateKey) {
    if m, e := ReadStream(c); e == nil {
      if m, e := Decrypt(k, m, []byte("served")); e == nil {
        Println(string(m))
      }
    }
  })
}

func LoadPrivateKey(file string) (r *rsa.PrivateKey, e error) {
  if file, e := ioutil.ReadFile(file); e == nil {
    if block, _ := pem.Decode(file); block != nil {
      if block.Type == "RSA PRIVATE KEY" {
        r, e = x509.ParsePKCS1PrivateKey(block.Bytes)
      }
    }
  }
  return
}
```

```
func Request(a, file string, f func(*UDPConn, *PrivateKey)) {
  if k, e := LoadPrivateKey(file); e == nil {
    if address, e := ResolveUDPAddr("udp", a); e == nil {
      if conn, e := DialUDP("udp", nil, address); e == nil {
        defer conn.Close()
        SendKey(conn, k.PublicKey, func() {
          f(conn, k)
        })
      }
    }
  }
}


func Decrypt(key *rsa.PrivateKey, m, l []byte) ([]byte, error) {
  return rsa.DecryptOAEP(sha1.New(), rand.Reader, key, m, l)
}


func SendKey(c *net.UDPConn, k rsa.PublicKey, f func()) {
  var b bytes.Buffer
  if e := gob.NewEncoder(&b).Encode(k); e == nil {
    if _, e = c.Write(b.Bytes()); e == nil {
      f()
    }
  }
}
```

```
package main

import "crypto/rsa"
import "crypto/rand"
import "crypto/sha1"
import "crypto/x509"
import "bytes"
import "encoding/gob"
import "encoding/pem"
import “io/ioutil"
import . "fmt"
import . "net"


func main() {
  Request(":1025", "ckey", func(c *net.UDPConn, k *rsa.PrivateKey) {
    if m, e := ReadStream(c); e == nil {
      if m, e := Decrypt(k, m, []byte("served")); e == nil {
        Println(string(m))
      }
    }
  })
}

func LoadPrivateKey(file string) (r *rsa.PrivateKey, e error) {
  if file, e := ioutil.ReadFile(file); e == nil {
    if block, _ := pem.Decode(file); block != nil {
      if block.Type == "RSA PRIVATE KEY" {
        r, e = x509.ParsePKCS1PrivateKey(block.Bytes)
      }
    }
  }
  return
}
```

```
func Request(a, file string, f func(*UDPConn, *PrivateKey)) {
  if k, e := LoadPrivateKey(file); e == nil {
    if address, e := ResolveUDPAddr("udp", a); e == nil {
      if conn, e := DialUDP("udp", nil, address); e == nil {
        defer conn.Close()
        SendKey(conn, k.PublicKey, func() {
          f(conn, k)
        })
      }
    }
  }
}


func Decrypt(key *rsa.PrivateKey, m, l []byte) ([]byte, error) {
  return rsa.DecryptOAEP(sha1.New(), rand.Reader, key, m, l)
}


func SendKey(c *net.UDPConn, k rsa.PublicKey, f func()) {
  var b bytes.Buffer
  if e := gob.NewEncoder(&b).Encode(k); e == nil {
    if _, e = c.Write(b.Bytes()); e == nil {
      f()
    }
  }
}
```