LONDON
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2015

goto; conference

Whispered Secrets

Eleanor McHugh

Join the conversation #gotoldn

Workshops: Sept 14-15 // Conference: Sept 16-18, 2015

Whispered Secrets

@feyeleanor

```go
package main
import (
  . "fmt"
  . "net/http"
  "sync"
)

const ADDRESS = ":1024"
const SECURE_ADDRESS = ":1025"

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

  var servers sync.WaitGroup
  servers.Add(1)
  go func() {
    defer servers.Done()
    ListenAndServe(ADDRESS, nil)
  }()

  servers.Add(1)
  go func() {
    defer servers.Done()
    ListenAndServeTLS(SECURE_ADDRESS, "cert.pem", "key.pem", nil)
  }()
  servers.Wait()
}
```

A Go Developer's Notebook

Eleanor McHugh

http://leanpub.com/GoNotebook

we all have secrets

and these secrets matter to us

that’s what makes them secrets

software should keep our secrets

some secrets are awful

conspiracy

infidelity

criminality

some secrets are banal

bank account numbers

embarrassing incidents

sexual preferences

secrecy should be absolute

our tech must protect the awful

or it won’t protect the banal

but there are laws

we must comply with these

assist the legitimate

deny the illegitimate

secrecy ——> privacy

privacy is not absolute

privacy requires mutual trust

mutual trust is a contract

and contracts can be broken

famous broken contracts

Ashley-Madison

Carphone Warehouse

Office of Personnel Management

today’s topic is applied paranoia

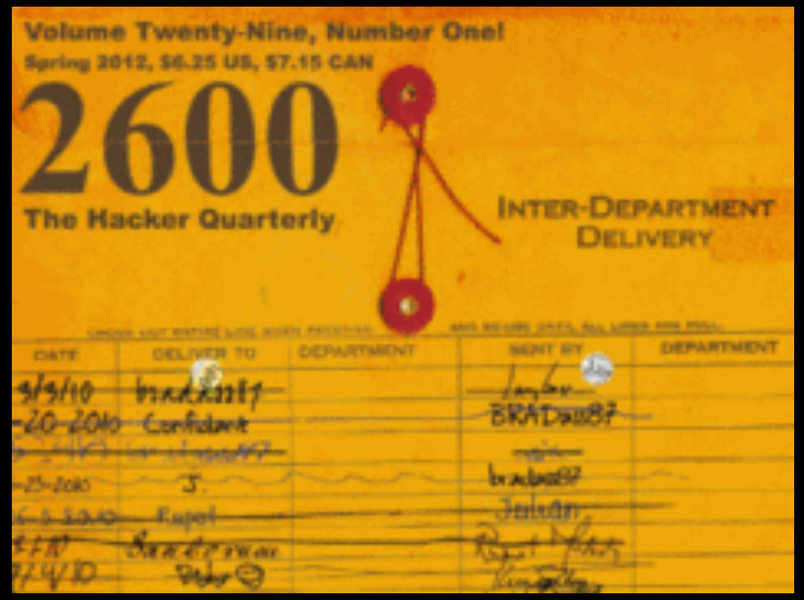
paranoia

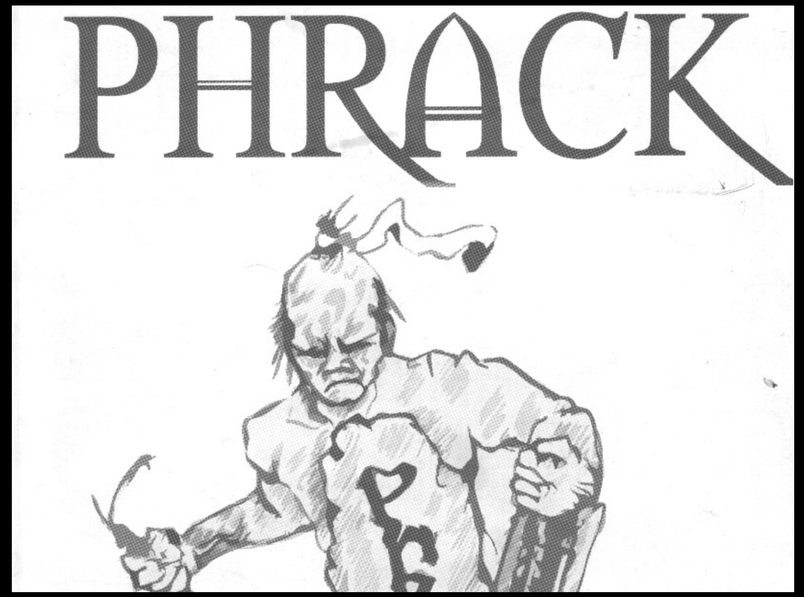
Pronunciation: /ˌparəˈnɔɪə/

noun
{mass noun}

A mental condition characterized by delusions of persecution, unwarranted jealousy, or exaggerated self-importance, typically worked into an organized system. It may be an aspect of chronic personality disorder, of drug abuse, or of a serious condition such as schizophrenia in which the person loses touch with reality.

Unjustified suspicion and mistrust of other people:
mild paranoia afflicts all prime ministers


Volume Twenty-Nine, Number One!
Spring 2012, $6.25 US, $7.15 CAN
2600
The Hacker Quarterly
INTER-DEPARTMENT DELIVERY
DATE
DELIVER TO
DEPARTMENT
SENT BY
DEPARTMENT


PHRACK

NATIONAL SECURITY AGENCY
UNITED STATES OF AMERICA

Daily Mail
MONDAY, MARCH 30, 2015
www.dailymail.co.uk
60p

On eve of pensions revolution, an exposé that will horrify every family in the land

YOUR PENSION SECRETS SOLD TO CONMEN FOR 5 PENCE

Daily Mail
INVESTIGATIONS UNIT
Katherine Faulkner, Paul Bentley and Lucy Osborne

HIGHLY sensitive details of the pension pots of millions are being sold for as little as 5p and ending up in the hands of criminals.

A Mail investigation reveals today how private financial information is being passed on by firms without their customers' knowledge.

This valuable data is then repeatedly sold on, ending up in the hands of fraudsters and cold calling firms.

The troubling [illegible] come on the eve of major government reforms that will hand mi[illegible]ons the chance to cash in their pen[illegible] pots - giving them access to huge [illegible] previously locked away. They will renew fears the reforms could trigger a flood of scams on the elderly and vulnerable whose newly-available funds will be rich pickings.

Last night the Information Commissioner's Office (ICO) vowed to pass evidence uncovered by the Mail to police and began an immediate inquiry into the firms involved. A Cabinet minister praised the investigation and

Turn to [illegible]

Daily Mail
TUESDAY, MARCH 31, 2015
www.dailymail.co.uk
60p

After Mail exposes trade in sensitive pension details...

NOW THEY ARE SELLING YOUR HEALTH SECRETS

Daily Mail
INVESTIGATIONS UNIT
Katherine Faulkner, Paul Bentley and Rosie Taylor

MEDICAL details on thousands of people are ending up in the hands of fraudsters and criminals after being secretly sold, a Mail investigation has revealed.

The details on sick and disabled people are being cynically touted by data firms for as little as 19p each.

They are being sold on with no checks to cold callers and fraudsters, who are often looking to target those who are at their most vulnerable.

The lists - often compiled from health insurance applications - contain details of thousands suffering with diabetes, high blood [illegible] osteoporosis, back pain and arthritis.

They name even those suffering from embarrassing bladder problems - as well as the frail of hearing, who can be especially vulnerable to scams.

The information obtained by the Mail was sold by a firm called Data Bubble owned by hypnotherapist Joanna Clayton. It is just one of hundreds of data brokers selling private information.

The latest revelations were branded 'abhorrent' and 'disgusting' by charities and MPs - with a number of them calling for those who sold such information to be jailed.

The news that medical data was being offered for sale comes after the Mail revealed how confidential details of savers' pensions were being sold

Turn to Page 4

Caught on camera: Joanna Clay[illegible]

paranoia

Pronunciation: /ˌparəˈnɔɪə/

noun
{*mass noun*}

The perfectly reasonable belief that someone, somewhere is watching your online behaviour with malicious and/or voyeuristic intent. It may be a result of reading a *Hacking Exposed* or *Hacking for Dummies* publication, experiencing the fallout from identity theft, or shopping with *bitcoin*.

Justified suspicion and mistrust of other people:

chronic paranoia afflicts all information security professionals
accute paranoia afflicts the victims of hacking


Check
AB E


EUROPEAN UNION
UNITED KINGDOM OF GREAT BRITAIN AND NORTHERN IRELAND
DIEU ET MON DROIT
DRIVING LICENCE
UK
An Executive Agency of the Home Office
Reference Number
Mr Andrew Carr
Biometrics
BIRTH
CHILD
OF AN ENTRY
Deaths Registration Act 1953
Administrative area
Entry No.
Male
Criminal Intelligence Service
INFORMAN

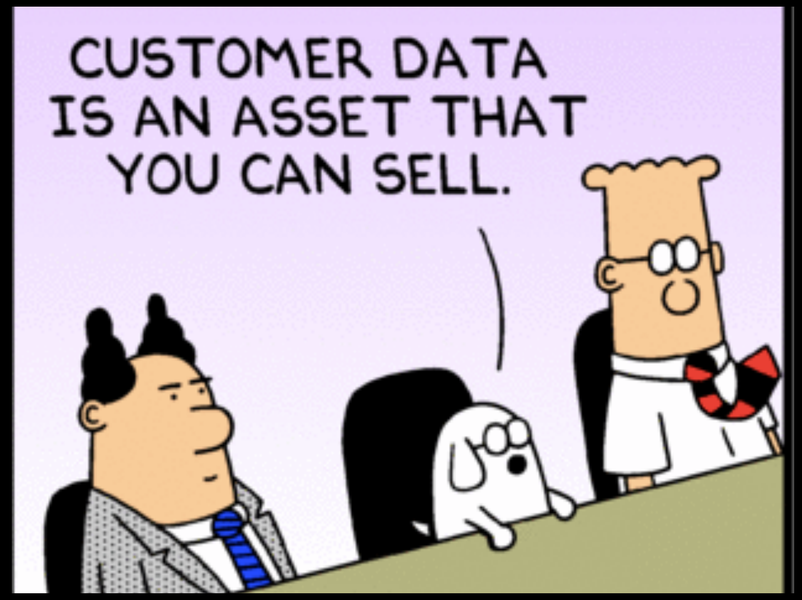
ny A8FWD3

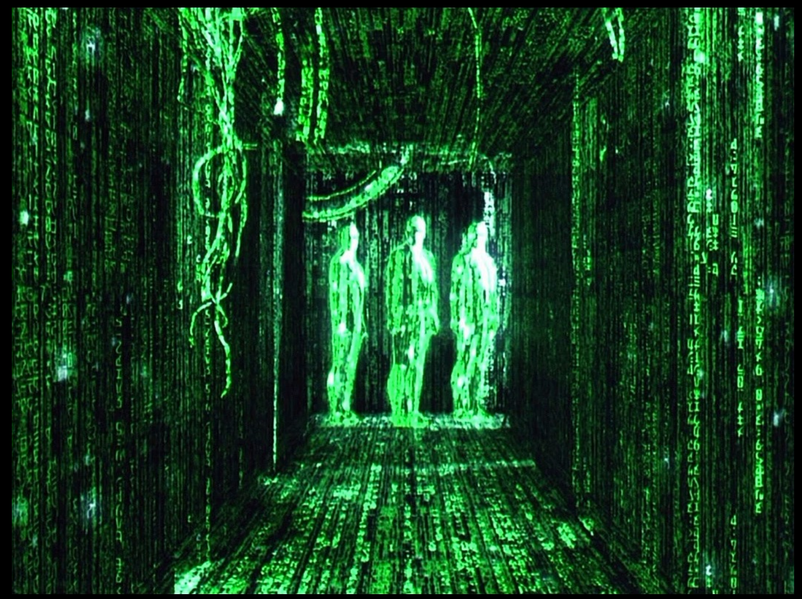
we have to trust governments

governments are privileged

if we don’t obey they can hurt us

not much we can do about that


CUSTOMER DATA IS AN ASSET THAT YOU CAN SELL.



our users have to trust us

our services are privileged

they store real-world secrets

and identifying metadata

but who can we trust?

technology bars the gates

but people create the bars

and people have to monitor them

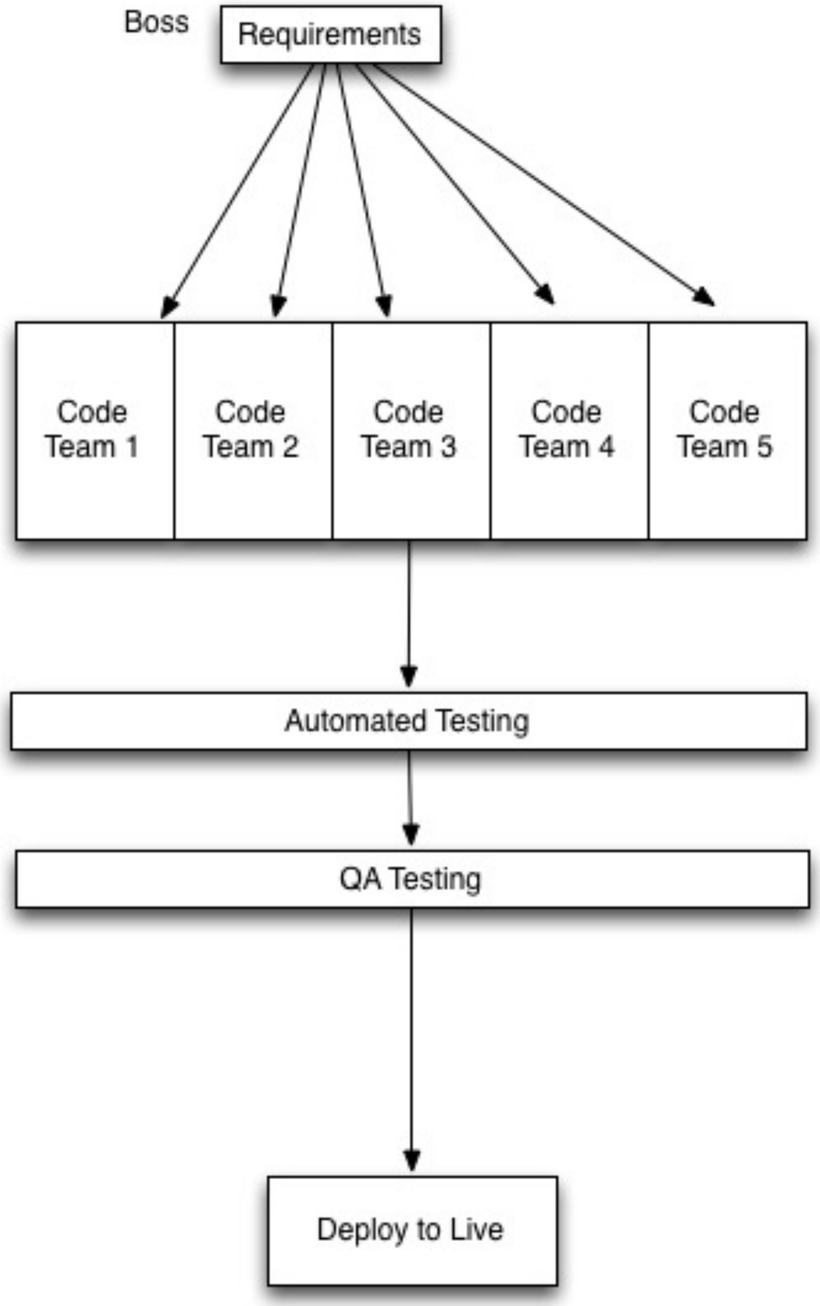
so what do we do?

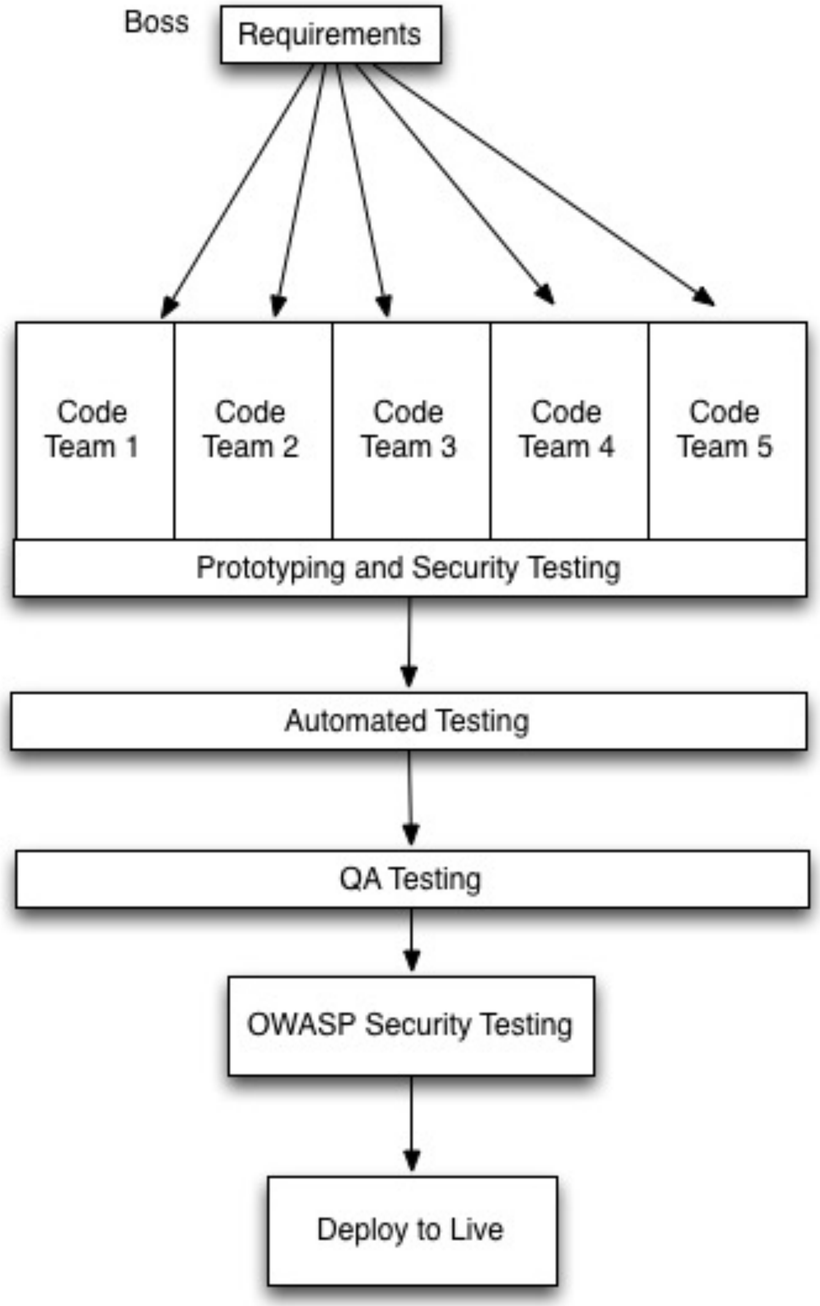
dev practices

architecture

operational rules

privacy ——> dev practices


Boss
Requirements
Code Team 1
Code Team 2
Code Team 3
Code Team 4
Code Team 5
Automated Testing
QA Testing
Deploy to Live


Boss
Requirements
Code Team 1
Code Team 2
Code Team 3
Code Team 4
Code Team 5
Prototyping and Security Testing
Automated Testing
QA Testing
OWASP Security Testing
Deploy to Live

privacy ——> architecture

encrypt all transports

- establish a secure channel by exchanging public keys
- and check their validity against trusted certificates (SSL, TLS, etc.)
- as an added measure pin these certificates (like SSH pins keys)
- then exchange symmetric keys for a private secure channel
- change these keys frequently (cheap cipher streams)
- and pin each distinct message to a distinct key (one-time pads)

https

```go
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)
}
```

```go
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)
}
```

```go
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

```go
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))
      }
    }
  }
}
```

```go
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
}
```

```go
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))
      }
    }
  }
}
```

```go
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
}
```

```go
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))
      }
    }
  }
}
```

```go
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

```go
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
}
```

```go
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)
  }
}
```

```go
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
}
```

```go
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)
  }
}
```

```go
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
}
```

```go
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

```go
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
}
```

```go
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
}
```

```go
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
}
```

```go
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
}
```

```go
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
}
```

```go
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
}
```

```go
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
}
```

```go
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
}
```

```go
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
}
```

```go
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

```go
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
}
```

```go
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)
    }
  }
}
```

```go
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
}
```

```go
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)
    }
  }
}
```

```go
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
}
```

```go
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)
    }
  }
}
```

```go
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
}
```

```go
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)
    }
  }
}
```

```go
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
}
```

```go
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

```go
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)
}
```

```go
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
}
```

```go
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)
}
```

```go
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

```go
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
}
```

```go
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()
    }
  }
}
```

```go
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
}
```

```go
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()
    }
  }
}
```

```go
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
}
```

```go
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()
    }
  }
}
```

aes + rsa —> hybrid crypto

aes + hmac —> signature

crypto + signature —> trust

hmac/rsa signing

```go
package main

import . "bytes"
import "crypto/hmac"
import "crypto/rsa"
import "crypto/sha256"
import "encoding/base64"
import "encoding/gob"
import "net"


func main() {
  HELLO_WORLD := []byte("Hello World")
  RSA_LABEL := []byte("served")
  SIGNING_KEY := []byte("signature")
  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 {
        m = append(Sign(HELLO_WORLD, SIGNING_KEY), m)
        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)
}
```

```go
func Sign(message string, key []byte) string {
    h := hmac.New(sha256.New, key)
    h.Write([]byte(message))
    return base64.StdEncoding.EncodeToString(h.Sum(nil))
}


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
}
```

```go
package main

import . "bytes"
import "crypto/hmac"
import "crypto/rsa"
import "crypto/sha256"
import "encoding/base64"
import "encoding/gob"
import "net"


func main() {
  HELLO_WORLD := []byte("Hello World")
  RSA_LABEL := []byte("served")
  SIGNING_KEY := []byte("signature")
  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 {
        m = append(Sign(HELLO_WORLD, SIGNING_KEY), m)
        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)
}
```

```go
func Sign(message string, key []byte) string {
    h := hmac.New(sha256.New, key)
    h.Write([]byte(message))
    return base64.StdEncoding.EncodeToString(h.Sum(nil))
}


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
}
```

hmac/rsa validation

```go
package main

import "crypto/hmac"
import "crypto/rsa"
import "crypto/rand"
import "crypto/sha1"
import "crypto/sha256"
import "crypto/x509"
import "bytes"
import "encoding/base64"
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[44:], []byte("served")); e == nil {
        s, _ := base64.URLEncoding.DecodeString(string(m[:44]))
        v := hmac.Equal(s, Sign(m, "signature"))
        Print(string(m), "[valid:", v,"]”)
      }
    }
  })
}

func Sign(message string, key []byte) string {
    h := hmac.New(sha256.New, key)
    h.Write([]byte(message))
    return base64.StdEncoding.EncodeToString(h.Sum(nil))
}
```

```go
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()
    }
  }
}
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)
}
```

```go
package main

import "crypto/hmac"
import "crypto/rsa"
import "crypto/rand"
import "crypto/sha1"
import "crypto/sha256"
import "crypto/x509"
import "bytes"
import "encoding/base64"
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[44:], []byte("served")); e == nil {
        s, _ := base64.URLEncoding.DecodeString(string(m[:44]))
        v := hmac.Equal(s, Sign(m, "signature"))
        Print(string(m), "[valid:", v,"]”)
      }
    }
  })
}

func Sign(message string, key []byte) string {
    h := hmac.New(sha256.New, key)
    h.Write([]byte(message))
    return base64.StdEncoding.EncodeToString(h.Sum(nil))
}
```

```go
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()
    }
  }
}
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)
}
```

encrypt all passwords

- accept unicode to expand the symbol space
- hash every new password **before** it's submitted
- always use a cryptograpically secure hash (HMAC)
- and a fresh HMAC key for each password (which you must store)
- salt the resulting hash when you receive it (and store the salt)
- then hash again before storing in your database

require multi-factor authentication

- have the user submit their password over a secure channel
- then send them a confirmation code out-of-band
- that’s an agreed trust anchor acting as a shared secret
- the confirmation code should be big enough to generate a HMAC
- and only the HMAC should be submitted
- now you have two secure channels based on shared secrets

encrypt all storage

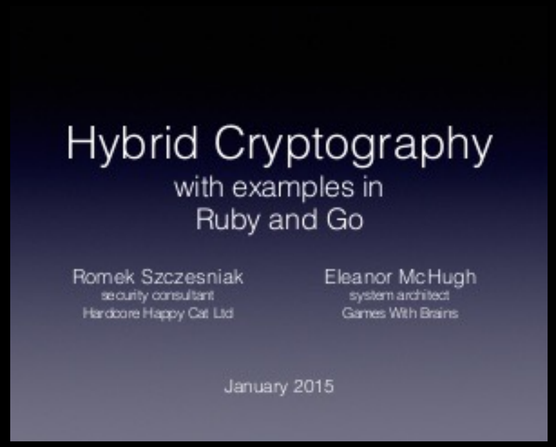
- secured transport is useless without secured data stores
- encrypt all sensitive fields - that probably means **all** fields
- and store HMACs for desired search terms
- otherwise your black box is secure but unsearchable
- make sure you use different roles for reading, writing and searching
- that’s right, your datastore is also a set of secure streams

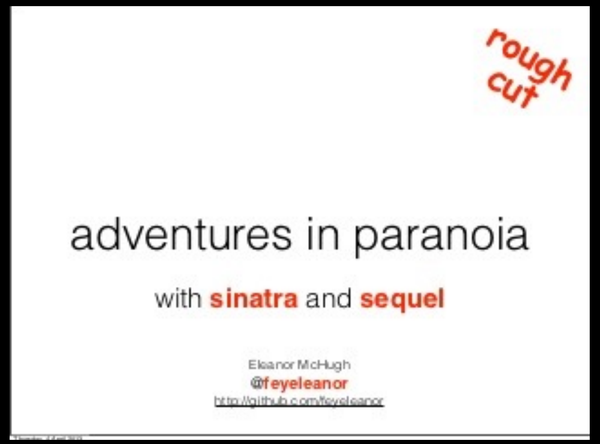
privacy ——> operational rules

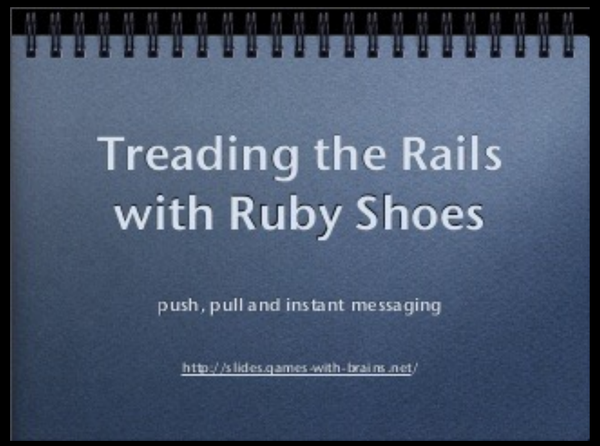
anchor trust internally

- establish a private certificate authority
- assign fine-grained roles to different components
- audit requirements, code, operations, security logs
- never deploy without a credible security audit
- and make those deployments immutable
- security audits best done by third parties with an attacker mentality

slideshare://feyeleanor







goto; conference

Click ‘engage’ to rate session.

Rate **12** sessions to get the supercool GOTO reward

Join the conversation #gotoldn