Crypto for Go Developers

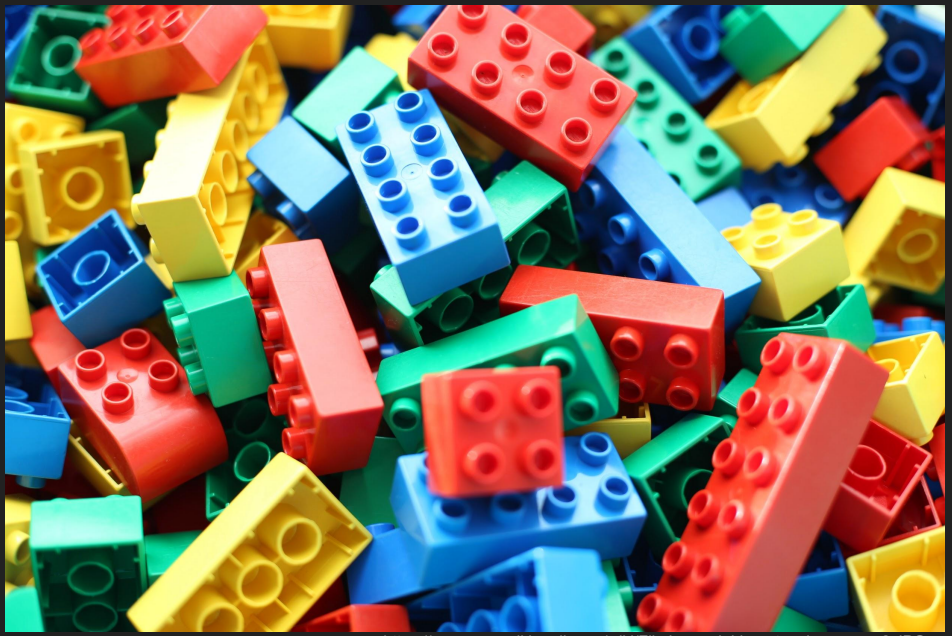
George Tankersley (@_gtank)
CoreOS

Don’t write your own crypto

This is Daniel J. Bernstein.
He designs the algorithms.

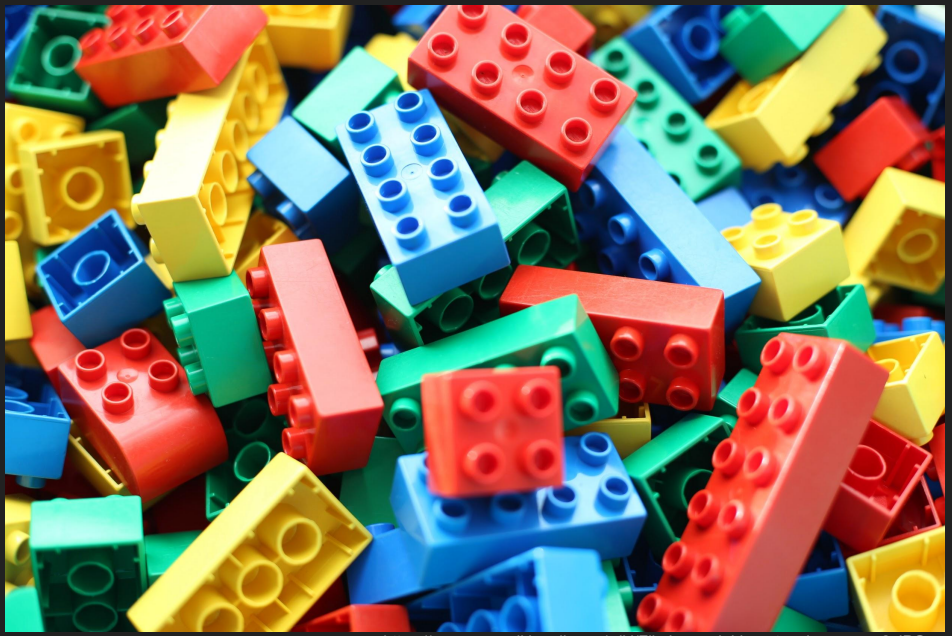
You don’t need to write your own crypto

Building blocks



https://commons.wikimedia.org/wiki/File:Lego_dublo_arto_alanenpaa_2.JPG

Building blocks



https://commons.wikimedia.org/wiki/File:Lego_dublo_arto_alanenpaa_2.JPG

TLS for data in motion
GPG for data at rest

How to use TLS

As a client

```
var minimalTLSConfig = &tls.Config{
     MinVersion: tls.VersionTLS12,
}

var tlsTransport = &http.Transport{
     TLSClientConfig: minimalTLSConfig,
}

var httpClient = &http.Client{
     Transport: tlsTransport,
     Timeout:   10 * time.Second,
}

func MakeRequest() error {
     resp, err := httpClient.Get("https://www.google.com")
     if err != nil {
          return err
     }

     // have fun
}
```

How to use TLS

As a client

```
var minimalTLSConfig = &tls.Config{
     MinVersion: tls.VersionTLS12,
}

var tlsTransport = &http.Transport{
     TLSClientConfig: minimalTLSConfig,
}

var httpClient = &http.Client{
     Transport: tlsTransport,
     Timeout:   10 * time.Second,
}

func MakeRequest() error {
     resp, err := httpClient.Get("https://www.google.com")
     if err != nil {
          return err
     }

     // have fun
}
```

How to use TLS

As a server

```
var minimalTLSConfig = &tls.Config{
     MinVersion:               tls.VersionTLS12,
     PreferServerCipherSuites: true,
}

var srv = &http.Server{
     Addr:      "localhost:8080",
     TLSConfig: minimalTLSConfig,
}

func handleReq(w http.ResponseWriter, r *http.Request) {
     fmt.Fprintf(w, "Hello, world")
}

func main() {
     http.HandleFunc("/", handleReq)

     err := srv.ListenAndServeTLS("cert.pem","key.pem")
     if err != nil {
          log.Fatal(err)
     }
}
```

How to use TLS

As a server

```
var minimalTLSConfig = &tls.Config{
     MinVersion:               tls.VersionTLS12,
     PreferServerCipherSuites: true,
}

var srv = &http.Server{
     Addr:      "localhost:8080",
     TLSConfig: minimalTLSConfig,
}

func handleReq(w http.ResponseWriter, r *http.Request) {
     fmt.Fprintf(w, "Hello, world")
}

func main() {
     http.HandleFunc("/", handleReq)

     err := srv.ListenAndServeTLS("cert.pem","key.pem")
     if err != nil {
          log.Fatal(err)
     }
}
```

TLS for data in motion
GPG for data at rest

How to use GPG

Please don’t use GPG

```
$ man gpg
GPG(1)                  GNU Privacy Guard 1.4                  GPG(1)

NAME
       gpg - OpenPGP encryption and signing tool

SYNOPSIS
       gpg [--homedir dir] [--options file] [options]
command [args]

DESCRIPTION
       gpg is the OpenPGP only version of the GNU Privacy
Guard (GnuPG). It is a tool to provide digital encryption
and signing services using the OpenPGP standard. gpg
features complete key management and all bells and whistles
you can expect from a decent OpenPGP implementation.

       This is the standalone version of gpg. For desktop
use you should consider using gpg2 from the GnuPG-2 package
([On some platforms gpg2 is installed under the name gpg]).
```

How to use GPG

Please don’t use GPG

```
$ man gpg
GPG(1)                  GNU Privacy Guard 1.4                  GPG(1)

NAME
       gpg - OpenPGP encryption and signing tool

SYNOPSIS
       gpg [--homedir dir] [--options file] [options]
command [args]

DESCRIPTION
       gpg is the OpenPGP only version of the GNU Privacy
Guard (GnuPG). It is a tool to provide digital encryption
and signing services using the OpenPGP standard. gpg
features complete key management and all bells and whistles
you can expect from a decent OpenPGP implementation.

       This is the standalone version of gpg. For desktop
use you should consider using gpg2 from the GnuPG-2 package
([On some platforms gpg2 is installed under the name gpg]).
```

How to use GPG

Please don’t use GPG

```
$ man gpg | wc -l
3227

$ man gpg | wc -w
16721
```

This is not a talk about TLS and GPG

Everyday cryptography that isn’t TLS

Hashing files

Generating random IDs

API authentication

Password storage for websites

Signed / encrypted cookies

JWTs

Signing updates

Just because it’s in `crypto/`
doesn’t mean it’s good

The `crypto/` package is vast and full of legacy

Encryption

- DES
- 3DES
- RC4
- TEA
- XTEA
- Blowfish
- Twofish
- CAST5
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

Encryption

The `crypto/` package is vast and full of legacy

Encryption

- DES
- 3DES
- RC4
- TEA
- XTEA
- Blowfish
- Twofish
- CAST5
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- 3DES
- RC4
- TEA
- XTEA
- Blowfish
- Twofish
- CAST5
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- RC4
- TEA
- XTEA
- Blowfish
- Twofish
- CAST5
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- TEA
- XTEA
- Blowfish
- Twofish
- CAST5
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- XTEA
- Blowfish
- Twofish
- CAST5
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- Blowfish
- Twofish
- CAST5
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- ~~Blowfish~~
- Twofish
- CAST5
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- ~~Blowfish~~
- Twofish
- ~~CAST5~~
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- ~~Blowfish~~
- ~~Twofish~~
- ~~CAST5~~
- Salsa20
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- ~~Blowfish~~
- ~~Twofish~~
- ~~CAST5~~
- ~~Salsa20~~
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

How NOT to use AES

Recall: the purpose of encryption is to hide the content of your data.

```
import (
     "crypto/aes"
     “crypto/rand”
     "fmt"
)

func main() {
     buf := []byte("The quick brown fox jumps over the lazy
dog")

     // Generate highly secure random AES key
     key := make([]byte, 32)
     _, err := rand.Read(key)
     if err != nil {
          panic(err)
     }

     aesCipher, _ := aes.NewCipher(key)

     // Encrypt in-place.
     aesCipher.Encrypt(buf, buf)
     fmt.Printf("%s\n", buf)
}
```

How NOT to use AES

Recall: the purpose of encryption is to hide the content of your data.

```
import (
	"crypto/aes"
	“crypto/rand”
	"fmt"
)

func main() {
	buf := []byte("The quick brown fox jumps over the lazy dog")

	// Generate highly secure random AES key
	key := make([]byte, 32)
	_, err := rand.Read(key)
	if err != nil {
		panic(err)
	}

	aesCipher, _ := aes.NewCipher(key)

	// Encrypt in-place.
	aesCipher.Encrypt(buf, buf)
	fmt.Printf("%s\n", buf)
}
```

How NOT to use AES

Recall: the purpose of encryption is to hide the content of your data.

```
import (
     "crypto/aes"
     “crypto/rand”
     "fmt"
)

func main() {
     buf := []byte("The quick brown fox jumps over the lazy
dog")

     // Generate highly secure random AES key
     key := make([]byte, 32)
     _, err := rand.Read(key)
     if err != nil {
          panic(err)
     }

     aesCipher, _ := aes.NewCipher(key)

     // Encrypt in-place.
     aesCipher.Encrypt(buf, buf)
     fmt.Printf("%s\n", buf)
}
```

How NOT to use AES

Recall: the purpose of encryption is to hide the content of your data.

```
import (
     "crypto/aes"
     “crypto/rand”
     "fmt"
)

func main() {
     buf := []byte("The quick brown fox jumps over the lazy
dog")

     // Generate highly secure random AES key
     key := make([]byte, 32)
     _, err := rand.Read(key)
     if err != nil {
          panic(err)
     }

     aesCipher, _ := aes.NewCipher(key)

     // Encrypt in-place.
     aesCipher.Encrypt(buf, buf)
     fmt.Printf("%s\n", buf)
}
```

How NOT to use AES

Recall: the purpose of encryption is to hide the content of your data.

```
Input: The quick brown fox jumps over the lazy dog.

Expect: ?)
?0??G1e???BQ?ʁ?c?R+t5?6O?
?

$ go run encrypt.go
ws)???efox jumps over the lazy dog.
```

How NOT to use AES

Recall: the purpose of encryption is to hide the content of your data.

```
Input: The quick brown fox jumps over the lazy dog.

Expect: ?)
?0??G1e???BQ?ʁ?c?R+t5?6O?
?

$ go run encrypt.go
ws)???efox jumps over the lazy dog.
```

Never use a `cipher.Block` directly

Instead, use a block cipher mode

Block cipher modes

Go offers CBC, CFB, CTR, OFB, and GCM modes.

Their details are not important.

Only GCM provides authenticated encryption.

Block cipher modes

Go offers ~~CBC~~, CFB, CTR, OFB, and GCM modes.

Their details are not important.

Only GCM provides authenticated encryption.

Block cipher modes

Go offers ~~CBC~~, ~~CFB~~, CTR, OFB, and GCM modes.

Their details are not important.

Only GCM provides authenticated encryption.

Block cipher modes

Go offers ~~CBC~~, ~~CFB~~, ~~CTR~~, OFB, and GCM modes.

Their details are not important.

Only GCM provides authenticated encryption.

Block cipher modes

Go offers ~~CBC~~, ~~CFB~~, ~~CTR~~, ~~OFB~~, and GCM modes.

Their details are not important.

Only GCM provides authenticated encryption.

How to encrypt

Initialize the block cipher

Choose a block cipher mode

Generate a randomized nonce

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Encrypt(data []byte, key [32]byte) ([]byte, error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	nonce := make([]byte, gcm.NonceSize())
	_, err = rand.Read(nonce)
	if err != nil {
		return nil, err
	}
	return gcm.Seal(nonce, nonce, data, nil), nil
}
```

How to encrypt

Initialize the block cipher

Choose a block cipher mode

Generate a randomized nonce

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Encrypt(data []byte, key [32]byte) ([]byte, error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	nonce := make([]byte, gcm.NonceSize())
	_, err = rand.Read(nonce)
	if err != nil {
		return nil, err
	}
	return gcm.Seal(nonce, nonce, data, nil), nil
}
```

How to encrypt

Initialize the block cipher

Choose a block cipher mode

Generate a randomized nonce

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Encrypt(data []byte, key [32]byte) ([]byte, error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	nonce := make([]byte, gcm.NonceSize())
	_, err = rand.Read(nonce)
	if err != nil {
		return nil, err
	}
	return gcm.Seal(nonce, nonce, data, nil), nil
}
```

How to encrypt

Initialize the block cipher

Choose a block cipher mode

Generate a randomized nonce

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Encrypt(data []byte, key [32]byte) ([]byte, error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	nonce := make([]byte, gcm.NonceSize())
	_, err = rand.Read(nonce)
	if err != nil {
		return nil, err
	}
	return gcm.Seal(nonce, nonce, data, nil), nil
}
```

How to encrypt

Initialize the block cipher

Choose a block cipher mode

Generate a randomized nonce

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Encrypt(data []byte, key [32]byte) ([]byte, error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	nonce := make([]byte, gcm.NonceSize())
	_, err = rand.Read(nonce)
	if err != nil {
		return nil, err
	}
	return gcm.Seal(nonce, nonce, data, nil), nil
}
```

How to encrypt

Initialize the block cipher

Choose a block cipher mode

Generate a randomized nonce

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Encrypt(data []byte, key [32]byte) ([]byte, error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	nonce := make([]byte, gcm.NonceSize())
	_, err = rand.Read(nonce)
	if err != nil {
		return nil, err
	}
	return gcm.Seal(nonce, nonce, data, nil), nil
}
```

How to decrypt

Initialize the block cipher

Choose a block cipher mode

We stored the nonce at the beginning of the encrypted data.

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Decrypt(ciphertext []byte, key [32]byte) (plaintext []
byte, err error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	return gcm.Open(nil,
		ciphertext[:gcm.NonceSize()],
		ciphertext[gcm.NonceSize():],
		nil,
	)
}
```

How to decrypt

Initialize the block cipher

Choose a block cipher mode

We stored the nonce at the beginning of the encrypted data.

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Decrypt(ciphertext []byte, key [32]byte) (plaintext []
byte, err error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	return gcm.Open(nil,
		ciphertext[:gcm.NonceSize()],
		ciphertext[gcm.NonceSize():],
		nil,
	)
}
```

How to decrypt

Initialize the block cipher

Choose a block cipher mode

We stored the nonce at the beginning of the encrypted data.

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Decrypt(ciphertext []byte, key [32]byte) (plaintext []
byte, err error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	return gcm.Open(nil,
		ciphertext[:gcm.NonceSize()],
		ciphertext[gcm.NonceSize():],
		nil,
	)
}
```

How to decrypt

Initialize the block cipher

Choose a block cipher mode

We stored the nonce at the beginning of the encrypted data.

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
)

func Decrypt(ciphertext []byte, key [32]byte) (plaintext []
byte, err error) {
	block, err := aes.NewCipher(key[:])
	if err != nil {
		return nil, err
	}

	gcm, err := cipher.NewGCM(block)
	if err != nil {
		return nil, err
	}

	return gcm.Open(nil,
		ciphertext[:gcm.NonceSize()],
		ciphertext[gcm.NonceSize():],
		nil,
	)
}
```

Hashes

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- ~~Blowfish~~
- ~~Twofish~~
- ~~CAST5~~
- ~~Salsa20~~
- AES

Hashes

- MD4
- MD5
- RIPEMD160
- SHA1
- SHA2
- SHA3

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- ~~Blowfish~~
- ~~Twofish~~
- ~~CAST5~~
- ~~Salsa20~~
- AES

Hashes

- ~~MD4~~
- ~~MD5~~
- ~~RIPEMD160~~
- ~~SHA1~~
- SHA2
- ~~SHA3~~

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

A word about hashes

Using hash functions directly is fraught with peril

- Length extension
- Rainbow tables
- Small number of possibilities (phone numbers)
- Salt? Pepper?

Like encryption, we use a construction on top of the basic algorithm

Instead of hash you should think HMAC

How to hash data

```
import (
     "crypto/hmac"
     "crypto/sha512"
)

func Hash(tag string, data []byte) []byte {
     h := hmac.New(sha512.New512_256, []byte(tag))
     h.Write(data)
     return h.Sum(nil)
}

func ExampleHash() error {
        tag := "hashing file for storage key"
        contents, err := ioutil.ReadFile("testfile")
        if err != nil {
                Return error
        }
        digest := Hash(tag, contents)
        fmt.Println(hex.EncodeToString(digest))
}

// Output:
// 9f4c795d8ae5c207f19184ccebee6a606c1fdfe509c793614066d613580f03e1
```

How to hash data

```
import (
     "crypto/hmac"
     "crypto/sha512"
)

func Hash(tag string, data []byte) []byte {
     h := hmac.New(sha512.New512_256, []byte(tag))
     h.Write(data)
     return h.Sum(nil)
}

func ExampleHash() error {
        tag := "hashing file for storage key"
        contents, err := ioutil.ReadFile("testfile")
        if err != nil {
                Return error
        }
        digest := Hash(tag, contents)
        fmt.Println(hex.EncodeToString(digest))
}

// Output:
// 9f4c795d8ae5c207f19184ccebee6a606c1fdfe509c793614066d613580f03e1
```

How to hash data

```
import (
     "crypto/hmac"
     "crypto/sha512"
)

func Hash(tag string, data []byte) []byte {
     h := hmac.New(sha512.New512_256, []byte(tag))
     h.Write(data)
     return h.Sum(nil)
}

func ExampleHash() error {
        tag := "hashing file for storage key"
        contents, err := ioutil.ReadFile("testfile")
        if err != nil {
                Return error
        }
        digest := Hash(tag, contents)
        fmt.Println(hex.EncodeToString(digest))
}

// Output:
// 9f4c795d8ae5c207f19184ccebee6a606c1fdfe509c793614066d613580f03e1
```

How to hash data

You can treat output from different tags as independent hash functions!

```
fileDigest := Hash("fileNode", []byte("hello, world"))
metaDigest := Hash("metadataNode", []byte("hello, world"))
fmt.Printf("%x\n%x\n", fileDigest, metaDigest)

$ go run example.go
e7332dd5f5b8f6b5af2403677805acaeca4820a7e319afc8951823de3d5ff25e
fe8894f9b7c9f0111680a3f85d94929e6d3c6f1ac379aeea1992023d277a4555
```

Hashing passwords

How to hash passwords

Passwords are a completely different situation.

Never use SHA2 or HMAC for passwords.

Use bcrypt. Use bcrypt. Use bcrypt. Use bcrypt.

```
import (
     "golang.org/x/crypto/bcrypt"
)

func HashPassword(password []byte) ([]byte, error) {
     return bcrypt.GenerateFromPassword(password, 14)
}

func CheckPasswordHash(hash, password []byte) error {
     return bcrypt.CompareHashAndPassword(hash, password)
}

func Example() {
     myPassword := []byte("password")
     hashed, err := HashPassword(myPassword)
     if err != nil {
          return
     }
     fmt.Println(string(hashed))
}

// Output:
// $2a$14$pCOIhZBzlW7URPHjZ8AFqu2DjsJ1LapFZaHq3mDksYzrgP3q6p4OO
```

How to hash passwords

Passwords are a completely different situation.

Never use SHA2 or HMAC for passwords.

Use bcrypt. Use bcrypt. Use bcrypt. Use bcrypt.

```
import (
     "golang.org/x/crypto/bcrypt"
)

func HashPassword(password []byte) ([]byte, error) {
     return bcrypt.GenerateFromPassword(password, 14)
}

func CheckPasswordHash(hash, password []byte) error {
     return bcrypt.CompareHashAndPassword(hash, password)
}

func Example() {
     myPassword := []byte("password")
     hashed, err := HashPassword(myPassword)
     if err != nil {
          return
     }
     fmt.Println(string(hashed))
}

// Output:
// $2a$14$pCOIhZBzlW7URPHjZ8AFqu2DjsJ1LapFZaHq3mDksYzrgP3q6p4OO
```

How to hash passwords

Passwords are a completely different situation.

Never use SHA2 or HMAC for passwords.

Use bcrypt. Use bcrypt. Use bcrypt. Use bcrypt.

```
import (
     "golang.org/x/crypto/bcrypt"
)

func HashPassword(password []byte) ([]byte, error) {
     return bcrypt.GenerateFromPassword(password, 14)
}

func CheckPasswordHash(hash, password []byte) error {
     return bcrypt.CompareHashAndPassword(hash, password)
}

func Example() {
     myPassword := []byte("password")
     hashed, err := HashPassword(myPassword)
     if err != nil {
          return
     }
     fmt.Println(string(hashed))
}

// Output:
// $2a$14$pCOIhZBzlW7URPHjZ8AFqu2DjsJ1LapFZaHq3mDksYzrgP3q6p4OO
```

Signatures

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- ~~Blowfish~~
- ~~Twofish~~
- ~~CAST5~~
- ~~Salsa20~~
- AES

Hashes

- ~~MD4~~
- ~~MD5~~
- ~~RIPEMD160~~
- ~~SHA1~~
- SHA2
- ~~SHA3~~

Signatures

- RSA
  - PKCS1v15
  - PSS
- ECDSA
  - P256
  - P384
  - P521
- Ed25519

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- ~~Blowfish~~
- ~~Twofish~~
- ~~CAST5~~
- ~~Salsa20~~
- AES

Hashes

- ~~MD4~~
- ~~MD5~~
- ~~RIPEMD160~~
- ~~SHA1~~
- SHA2
- ~~SHA3~~

Signatures

- ~~RSA~~
  - ~~PKCS1v15~~
  - ~~PSS~~
- ECDSA
  - P256
  - ~~P384~~
  - ~~P521~~
- ~~Ed25519~~

Generate an ECDSA key

Math is happening

```
import (
     "crypto/ecdsa"
     "crypto/elliptic"
     "crypto/rand"
)

func NewSigningKey() (*ecdsa.PrivateKey, error) {
     key, err := ecdsa.GenerateKey(elliptic.P256(), rand.
Reader)
     return key, err
}
```

Generate an ECDSA key

Math is happening

```
import (
     "crypto/ecdsa"
     "crypto/elliptic"
     "crypto/rand"
)

func NewSigningKey() (*ecdsa.PrivateKey, error) {
     key, err := ecdsa.GenerateKey(elliptic.P256(), rand.
Reader)
     return key, err
}
```

How to sign data

Hash the data and sign the hash using the package-level `Sign`

For compatibility with JWTs, store signatures as a big-endian array of two large integers in R,S order

```
func Sign(data []byte, priv *ecdsa.PrivateKey) ([]byte,
error) {

     digest := sha256.Sum256(data)

     r, s, err := ecdsa.Sign(rand.Reader, priv, digest[:])
     if err != nil {
          return nil, err
     }

     // encode the signature {R, S}
     params := priv.Curve.Params()
     curveByteSize := params.P.BitLen() / 8
     rBytes, sBytes := r.Bytes(), s.Bytes()
     signature := make([]byte, curveByteSize*2)
     copy(signature[curveByteSize-len(rBytes):], rBytes)
     copy(signature[curveByteSize*2-len(sBytes):], sBytes)

     return signature, nil
}
```

How to sign data

Hash the data and sign the hash using the package-level `Sign`

For compatibility with JWTs, store signatures as a big-endian array of two large integers in R,S order

```
func Sign(data []byte, priv *ecdsa.PrivateKey) ([]byte,
error) {

    digest := sha256.Sum256(data)

    r, s, err := ecdsa.Sign(rand.Reader, priv, digest[:])
    if err != nil {
        return nil, err
    }

    // encode the signature {R, S}
    params := priv.Curve.Params()
    curveByteSize := params.P.BitLen() / 8
    rBytes, sBytes := r.Bytes(), s.Bytes()
    signature := make([]byte, curveByteSize*2)
    copy(signature[curveByteSize-len(rBytes):], rBytes)
    copy(signature[curveByteSize*2-len(sBytes):], sBytes)

    return signature, nil
}
```

How to sign data

Hash the data and sign the hash using the package-level `Sign`

For compatibility with JWTs, store signatures as a big-endian array of two large integers in R,S order

```
func Sign(data []byte, priv *ecdsa.PrivateKey) ([]byte,
error) {

     digest := sha256.Sum256(data)

     r, s, err := ecdsa.Sign(rand.Reader, priv, digest[:])
     if err != nil {
          return nil, err
     }

     // encode the signature {R, S}
     params := priv.Curve.Params()
     curveByteSize := params.P.BitLen() / 8
     rBytes, sBytes := r.Bytes(), s.Bytes()
     signature := make([]byte, curveByteSize*2)
     copy(signature[curveByteSize-len(rBytes):], rBytes)
     copy(signature[curveByteSize*2-len(sBytes):], sBytes)

     return signature, nil
}
```

How to verify data

Again, we have to hash the data

This is the same signature marshaling we just used.

```
import (
    "crypto/ecdsa"
    "crypto/sha256"
    "math/big"
)

// Returns true if it's valid and false if not.
func Verify(data, sig []byte, pub *ecdsa.PublicKey) bool {
    digest := sha256.Sum256(data)

    curveByteSize := pub.Curve.Params().P.BitLen() / 8
    r, s := new(big.Int), new(big.Int)
    r.SetBytes(signature[:curveByteSize])
    s.SetBytes(signature[curveByteSize:])

    return ecdsa.Verify(pub, digest[:], r, s)
}
```

How to verify data

Again, we have to hash the data

This is the same signature marshaling we just used.

```
import (
    "crypto/ecdsa"
    "crypto/sha256"
    "math/big"
)

// Returns true if it's valid and false if not.
func Verify(data, sig []byte, pub *ecdsa.PublicKey) bool {
    digest := sha256.Sum256(data)

    curveByteSize := pub.Curve.Params().P.BitLen() / 8
    r, s := new(big.Int), new(big.Int)
    r.SetBytes(signature[:curveByteSize])
    s.SetBytes(signature[curveByteSize:])

    return ecdsa.Verify(pub, digest[:], r, s)
}
```

How to verify data

Again, we have to hash the data

This is the same signature marshaling we just used.

```
import (
    "crypto/ecdsa"
    "crypto/sha256"
    "math/big"
)

// Returns true if it's valid and false if not.
func Verify(data, sig []byte, pub *ecdsa.PublicKey) bool {
    digest := sha256.Sum256(data)

    curveByteSize := pub.Curve.Params().P.BitLen() / 8
    r, s := new(big.Int), new(big.Int)
    r.SetBytes(signature[:curveByteSize])
    s.SetBytes(signature[curveByteSize:])

    return ecdsa.Verify(pub, digest[:], r, s)
}
```

The `crypto/` package is vast and full of legacy

Encryption

- ~~DES~~
- ~~3DES~~
- ~~RC4~~
- ~~TEA~~
- ~~XTEA~~
- ~~Blowfish~~
- ~~Twofish~~
- ~~CAST5~~
- ~~Salsa20~~
- AES

Hashes

- ~~MD4~~
- ~~MD5~~
- ~~RIPEMD160~~
- ~~SHA1~~
- SHA2
- ~~SHA3~~

Signatures

- ~~RSA~~
  - ~~PKCS1v15~~
  - ~~PSS~~
- ECDSA
  - P256
  - ~~P384~~
  - ~~P521~~
- ~~Ed25519~~

The `crypto/` package is vast and full of legacy

Encryption

- AES

Hashes

- SHA2

Signatures

- ECDSA
  - P256

Did you get all that?

Because there’s more!

Bonus: Random numbers

You want `crypto/rand`

As of Go 1.6, be careful with goimports!

```
import (
     "crypto/aes"
     "crypto/cipher"
     "crypto/rand"
     "io”
)

func DontDoThisAnymore() [32]byte {
     key := [32]byte{}
     _, err := rand.Read(key[:])
     if err != nil {
          panic(err)
     }
     return key
}

func NewEncryptionKey() [32]byte {
     key := [32]byte{}
     _, err := io.ReadFull(rand.Reader, key[:])
     if err != nil {
          panic(err)
     }
     return key
}
```

Bonus: Random numbers

You want `crypto/rand`

As of Go 1.6, be careful with goimports!

```
import (
	"crypto/aes"
	"crypto/cipher"
	"crypto/rand"
	"io”
)

func DontDoThisAnymore() [32]byte {
	key := [32]byte{}
	_, err := rand.Read(key[:])
	if err != nil {
		panic(err)
	}
	return key
}

func NewEncryptionKey() [32]byte {
	key := [32]byte{}
	_, err := io.ReadFull(rand.Reader, key[:])
	if err != nil {
		panic(err)
	}
	return key
}
```

Now available as a library (surprise!)

All of the stuff in the presentation, optimized for safe copy & paste

https://github.com/gtank/cryptopasta

Questions?