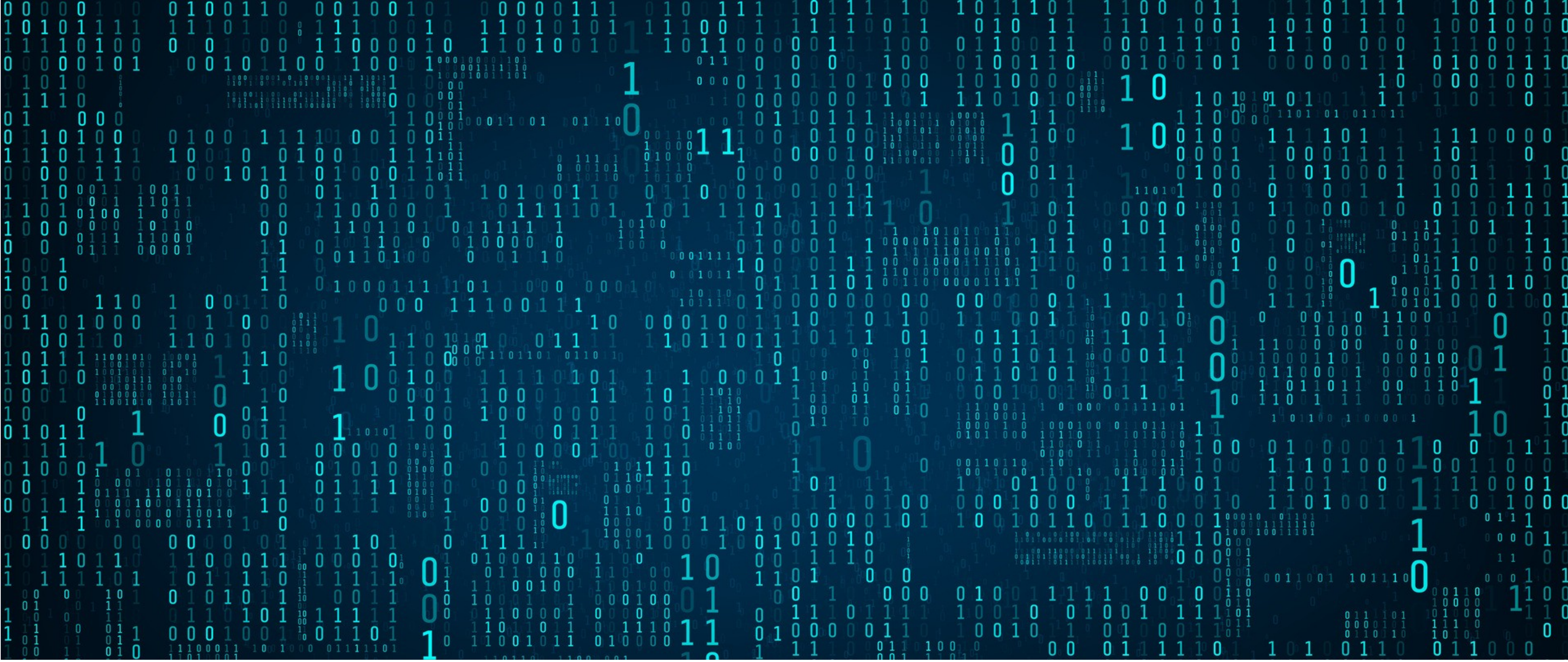


Advanced Cryptography in ColdFusion

Justin Scott, CISSP

Chief Information Security Officer, Smart Communications

Adobe ColdFusion Summit

October 1, 2024

Hi, I’m Justin Scott

- BBS sysop in the mid 1990’s
- Won a copy of Allaire ColdFusion 4 at SysCon in 1999
- Architect and developer for hundreds of applications
- Network, Systems, and Database admin
- Smart Communications since 2009 as IT Director, VP of Technology, and most recently Chief Information Security Officer
- Patent awarded as a co-inventor on a system for secure mail processing at correctional facilities
- CISSP

First, Common Questions

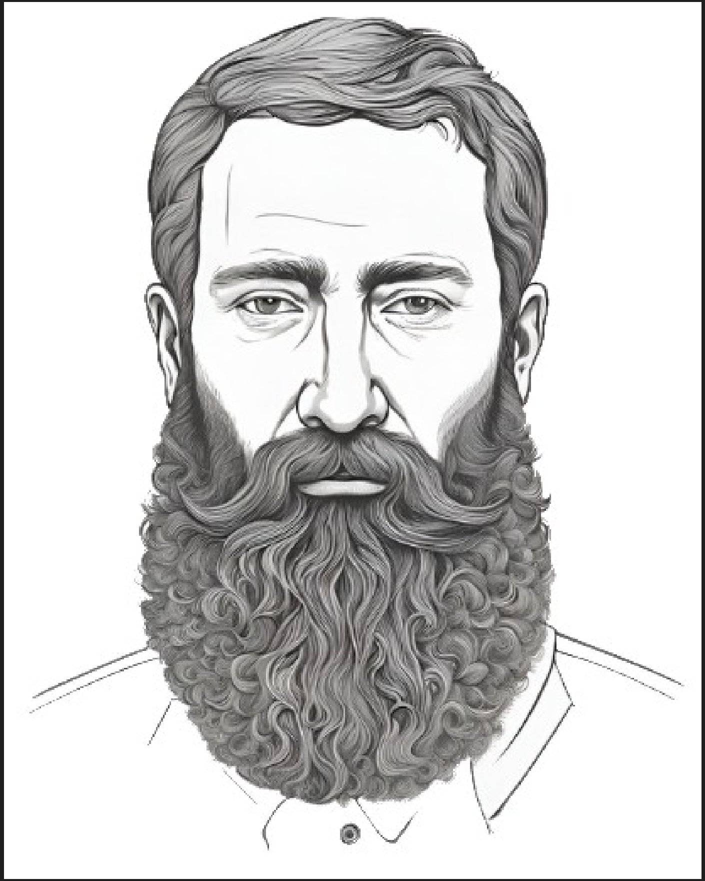
Three Years to Grow

No Special Method

(Just genetics)

No Special Routines

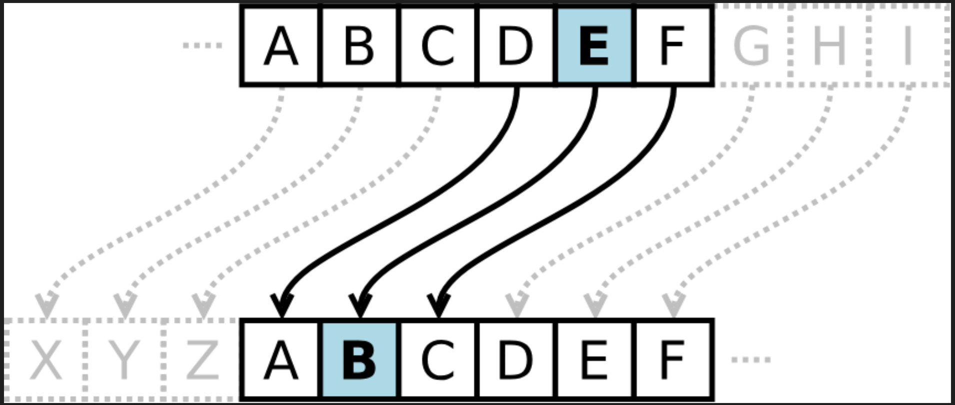
(I wake up like this)



Let’s Get TECHNICAL!

What is Cryptography?

The art and science of “secret writing”



Confidentiality

Ensuring that data cannot be read if exposed to the public, typically through a symmetric or asymmetric encryption algorithm.

Authenticity

Ensuring that the data came from a specific source, such as a digital signature on an email or code signing, typically by using public key infrastructure to verify the author.

Access Control

Ensuring that a user is properly authenticated and authorized to access data, typically through signed session tokens.

Integrity

Ensuring that data has not been tampered with, typically by applying a secure hashing algorithm to generate a one-way fingerprint of the data.

Non-Repudiation

Ensuring that a sender or receiver cannot deny that a message was sent or received, typically by using public key infrastructure to sign messages or read receipts.

Typical Uses of Cryptography

- Password Hashing
- Protect Sensitive/Private Information
- Store Documents
- Access Tokens

Hashing

Hash, from French culinary terminology meaning “to chop and mix.”

Takes data of arbitrary length and generates a fixed-length output.

Output can be used as a “fingerprint” of the input data.

Small changes to the input should generate vastly different output.

Good hashing algorithms avoid collisions.

Primarily used for integrity checks (e.g. has this data changed?).

Also known as creating a “digest” of the data.

Password Hashing

Best practice is to store passwords as a one-way hash using a secure hashing algorithm.

ColdFusion provides several functions to enable this common use case.

Ye Olden Times...

```
<cfquery name="getUser" datasource="#application.dsn#">
    SELECT userID, password
    FROM users
    WHERE username = '#form.username#'
</cfquery>

<cfif getUser.password is form.password>
    <cfset userValid = true>
</cfif>
```

A Slightly Better Approach

```
getUser = queryExecute("
    SELECT userID, password
    FROM users
    WHERE username = :user
", { user=form.username });

if (getUser.password is hash(form.password)) {
    userValid = true;
}
```

Password Hashing

Simple example:

```
hash("my1337p@ssword");
```

Output (pre CF 2023 update 8 or 2021 update 14 – CFMX_COMPAT [MD5*]):

```
E261DB47EFBA4DBEB805B7D4A73CD27E
```

Password Hashing

Simple example:

```
hash("my1337p@ssword");
```

Output (CF 2023 update 8 or 2021 update 14 and after – SHA-256):

```
FE49AF781B98D482B4D5DA40203D44573F12E4F5754A5A8B2CC85A7246074FDD
```

Password Hashing

We can also specify an algorithm:

```
hash("my1337p@ssword", "SHA-512");
```

Output:

```
4896FDA84CF5F079A32755579FE6195D538877E5B7055847F209131E63E05E4A80C
7446C6D6D2A733381F287DB8164531E437774FF01F60EBD16B3C524B030B8
```

Algorithm Options for `hash(“my1337p@ssword”,` *`algorithm`*`);`

MD2 (Output: 128bits, 16 bytes, 32 hex characters):
98F713E711782949C410951DF61A6F2D

MD5 / **CFMX_COMPAT*** (Output: 128bits, 16 bytes, 32 hex characters):
E261DB47EFBA4DBEB805B7D4A73CD27E

SHA / **SHA-1** (Output: 160bits, 20 bytes, 40 hex characters):
22997B8C8CBD8C682360CF064E87AA28DA6E7F8D

RIPEMD160 (Output: 160bits, 20 bytes, 40 hex characters):
F129DB8A056298A338EADAB814DF600DB18E2780

SHA-224 (Output: 224bits, 28 bytes, 56 hex characters):
0AA3CE433C3B506F4BCD5AF087A0F1ACAB83E528789B5DEB44A13B5F

SHA-256 (Output: 256bits, 32 bytes, 64 hex characters):
FE49AF781B98D482B4D5DA40203D44573F12E4F5754A5A8B2CC85A7246074FDD

SHA-384 (Output: 384bits, 48 bytes, 96 hex characters):
10E0DB52B0332ACBE3C4D00B58F24C01F11CE1902C41D03C28CF6F11B7129A40A47CCA7081A97893B143011E34776B72

SHA-512 (Output: 512bits, 64 bytes, 128 hex characters):
4896FDA84CF5F079A32755579FE6195D538877E5B7055847F209131E63E05E4A80C7446C6D6D2A733381F287DB8164531E4
37774FF01F60EBD16B3C524B030B8

Hashing Alone Isn’t Enough

We Need to Add Some SALT

Unique Salt Per Password

Giving every password a unique salt ensures that no two stored hashes will be the same even if the passwords are the same.

Use a lot of Salt

Salts should be long to avoid collisions.
Use a value of at least 128 bits (16 bytes).
Don’t use the user ID primary key from the database.

Some Options

```
createUUID();
generateSecretKey(“AES”, 128);
```

Salts Aren’t Secret

The salt can be stored in plain text in the database alongside the password hash.

New Password = New Salt

Every time the password is changed, generate a new salt value. Do not reuse salts.

Salted Hash Example

```
form.userPass = "my1337p@ssword";
variables.salt = createUUID(); // A7012479-AD02-F396-8863BE8C50F920BC
variables.pwHash = hash(form.userPass & variables.salt, "SHA-512");
```

Output (512bits, 64 bytes, 128 hex characters):

```
33C2EEAACED5F7D3E46E12EBBAF8B6EFB9F1898D59CDE373E5B5E2CBDD894E3
F69574B02415F1E721142054F91BD7A76D260E569592004B532820BE14C55CC9C
```

Now we can store the salt value and the password hash in the database. At login, repeat the hash using the password input and the stored salt for comparison.

We Need to Add a Work Factor

Hash with Iterations

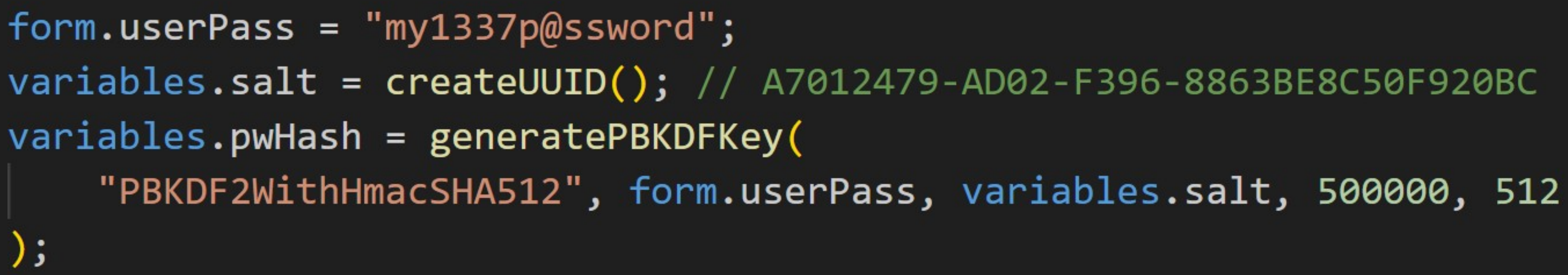
The hash() function also provides a parameter for additional iterations.

```
form.userPass = "my1337p@ssword";
variables.salt = createUUID(); // A7012479-AD02-F396-8863BE8C50F920BC
variables.pwHash = hash(
    form.userPass & variables.salt,
    "SHA-512",
    "UTF-8",
    1000000); // Iterations
```

This now takes longer; ~370ms on my test server

Password-Based Key Derivation

ColdFusion provides the generatePBKDFKey() function which does more work than a hash() alone, and also supports an iteration count.

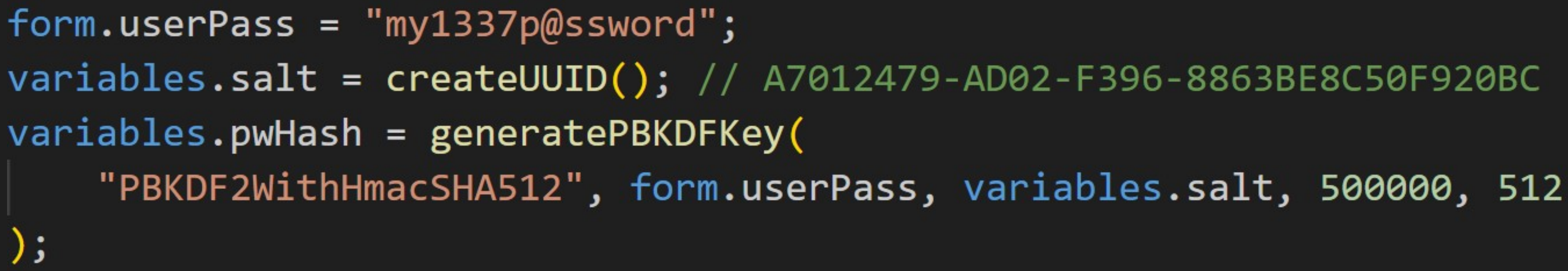


```
form.userPass = "my1337p@ssword";
variables.salt = createUUID(); // A7012479-AD02-F396-8863BE8C50F920BC
variables.pwHash = generatePBKDFKey(
    "PBKDF2WithHmacSHA512", form.userPass, variables.salt, 500000, 512
);
```

This takes even longer with half as many iterations; ~700ms on my server

Password-Based Key Derivation

Output is encoded using base64 vs. hex for hash().



```
form.userPass = "my1337p@ssword";
variables.salt = createUUID(); // A7012479-AD02-F396-8863BE8C50F920BC
variables.pwHash = generatePBKDFKey(
    "PBKDF2WithHmacSHA512", form.userPass, variables.salt, 500000, 512
);
```

Output:

```
Y40YYsMWxv7DGb681oL6/eQFsElHE78549ReKyPncP634ag878SYK0nRwSPaf3FN7R2o2x324fdIlJZD7dfeXQ==
```

Add Some PEPPER to That Hash

Peppered Hash Example

```
variables.pepper = "SOME_PEPPER"; // NOTE this should come from secrets manager
form.userPass = "my1337p@ssword";
variables.pepPass = form.userPass & variables.pepper;
variables.salt = createUUID(); // A7012479-AD02-F396-8863BE8C50F920BC
variables.pwHash = hash(variables.pepPass & variables.salt, "SHA-512");
```

Output (512bits, 64 bytes, 128 hex characters):

```
A9425D78CAF7FEF256AB75E7321DDC3AE29FC278668D4C9E95842E5F9CE8DBE05
D0C0A9FA8F033F18E16566B65BBAD11C82EC364562DA26EF61D87E3C0B91E99
```

BCrypt and SCrypt

A more modern approach to password hashing.

Introduced in ColdFusion 2021.

Include automatic hash generation and work factors.

Separate “generate” and “verify” functions.

Most parameters used to generate are included with the result, so updating code is easier as older hashes will still verify even if you increase the workload on newer hashes.

We don’t have to do it all manually with hash() anymore!

BCrypt Example

```
variables.pepper = "SOME_PEPPER";
form.userPass = "my1337p@ssword";
variables.pepPass = form.userPass & variables.pepper;
options = { version: "$2a", rounds: 10 };
variables.pwHash = generateBCryptHash(variables.pepPass, options);
```

Output (fixed-length 60 character string):

$2a$10$xlyn28s5RcvIjeaQmwpRIemzfVy4kgsTEfME2JWdfDLw8E/msCHRm

Includes the BCrypt version, work factor, 22 character base64 encoded salt, and the 31 character base64 encoded password hash.

BCrypt Example

```
variables.pepper = "SOME_PEPPER";
form.userPass = "my1337p@ssword";
variables.pepPass = form.userPass & variables.pepper;
options = { version: "$2a", rounds: 10 };
variables.pwHash = generateBCryptHash(variables.pepPass, options);
```

Output (fixed-length 60 character string):

```
$2a$10$xlyn28s5RcvIjeaQmwpRIemzfVy4kgsTEfME2JWdfDLw8E/msCHRm
```

Includes the BCrypt version, work factor, 22 character base64 encoded salt, and the 31 character base64 encoded password hash.

BCrypt Example

```
getUser.pwHash =
    "$2b$12$IG8HdilP2Y3oNm09MRq6z.ZjtpGNiQw8V/LWrjwjEzjIpgsTbthB2";
variables.pepper = "SOME_PEPPER";
form.userPass = "my1337p@ssword";
variables.pepPass = form.userPass & variables.pepper;
variables.isValid = verifyBCryptHash(variables.pepPass, getUser.pwHash);

// isValid == YES
```

Options are included on the stored hash, so no need to pass them into the verify function on their own.

Verification takes the same amount of time as generation.

SCrypt Example

```
variables.pepper = "SOME_PEPPER";
form.userPass = "my1337p@ssword";
variables.pepPass = form.userPass & variables.pepper;
options = { cpucost=16384, memorycost=8, keylength=32, saltlength=8 };
variables.pwHash = generateSCryptHash(variables.pepPass, options);
```

Output (variable-length string depending on salt length and key length):

$e0801$84olBR5KPNY=$mPh3FFNHUGCpt5vHa+YusCnwOTsQdSzoftlGmOt3Hwo=

Includes the input parameters, base64-encoded salt, and the resulting hash.

Note: Output is supposed to include a version string at the beginning ($s0), but the CF function does not include this, so it needs to be added manually for interoperability with other libraries.

SCrypt Example

```
getUser.pwHash =
    "$e0801$84olBR5KPNY=$mPh3FFNHUGCpt5vHa+YusCnwOTsQdSzoftlGmOt3Hwo=";
variables.pepper = "SOME_PEPPER";
form.userPass = "my1337p@ssword";
variables.pepPass = form.userPass & variables.pepper;
variables.isValid = verifySCryptHash(variables.pepPass, getUser.pwHash);

// isValid == YES
```

Options are included on the stored hash, so no need to pass them into the verify function on their own.

Note: If the original hash came from another application, you will need to strip off the leading $s0 version identifier before verification.

Argon2id Would be Even Better

Passkeys are Coming…

They use asymmetric encryption with public and private keys and are far more secure than passwords, but implementation is a mess right now.

Symmetric Key Encryption

Asymmetric Key Encryption

ColdFusion Encryption Functions

encrypt()
encryptBinary()
decrypt()
decryptBinary()
generateSecretKey()
generatePBKDFKey()

encrypt() Example

```
variables.key = "super secret key";
variables.plainText = "Secret String";
variables.cipherText = encrypt(plainText, key);
```

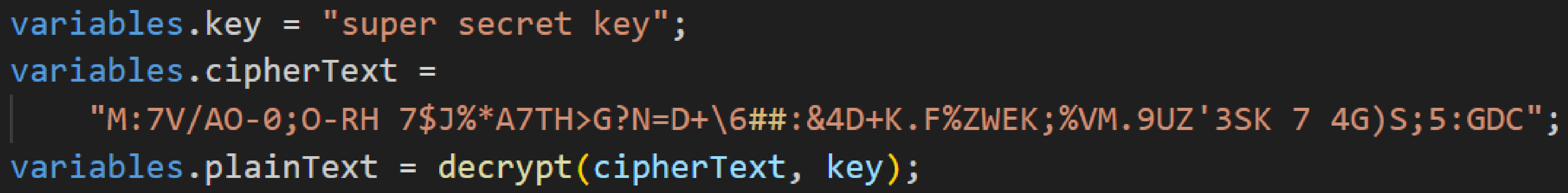
Output (pre-update “CFMX_COMPAT”, “UU” encoding):

```
M:7V/AO-0;O-RH 7$J%*A7TH>G?N=D+\6#:&4D+K.F%ZWEK;%VM.9UZ'3SK 7 4G)S;5:GDC
```

CFMX_COMPAT (pre-update default) “...uses an XOR-based algorithm that uses a pseudo-random 32-bit key, based on a seed passed by the user as a function parameter.”

Not very secure and should not be used in production.

decrypt() Example



```
variables.key = "super secret key";
variables.cipherText =
    "M:7V/AO-0;O-RH 7$J%*A7TH>G?N=D+\6##:&4D+K.F%ZWEK;%VM.9UZ'3SK 7 4G)S;5:GDC";
variables.plainText = decrypt(cipherText, key);
```

Output:

```
Secret String
```

Algorithm Options

Like hash() we can specify an algorithm. Options include:

CFMX_COMPAT – Basic XOR algorithm that uses a string seed to generate a 33bit key. Weak.

DES – Data Encryption Standard (56 bit keys; can be brute forced in several days)

DESEDE – Triple DES (112 or 168 bit keys; 64 bit blocks; deprecated by NIST in 2019)

BLOWFISH – Defined by Bruce Schneier in 1993 (32-448 bit keys; 64 bit blocks)

AES – Advanced Encryption Standard (formerly Rijndael; 128, 192, and 256 bit keys; 128 bit blocks)

Also: RC2, RC4, RC5, PBE, DESX – Don’t use these unless needed for compatibility.

In Short… Use AES

(Preferably with a 256 bit key for quantum readiness)

encrypt() Example with AES

```
variables.key = generateSecretKey("AES"); // u8VVxZV9saMt7vUir5L+og== (128bit)
variables.plainText = "Secret String";
variables.cipherText = encrypt(plainText, key, "AES", "base64");
```

Output:

```
wP+cd7Dk+I3RRSB56t1DOw==
```

The AES Algorithm

AES performs a series of transformation rounds (10, 12, or 14 rounds depending on key size) to encrypt the 128 bits of input data. Each round involves four key steps:

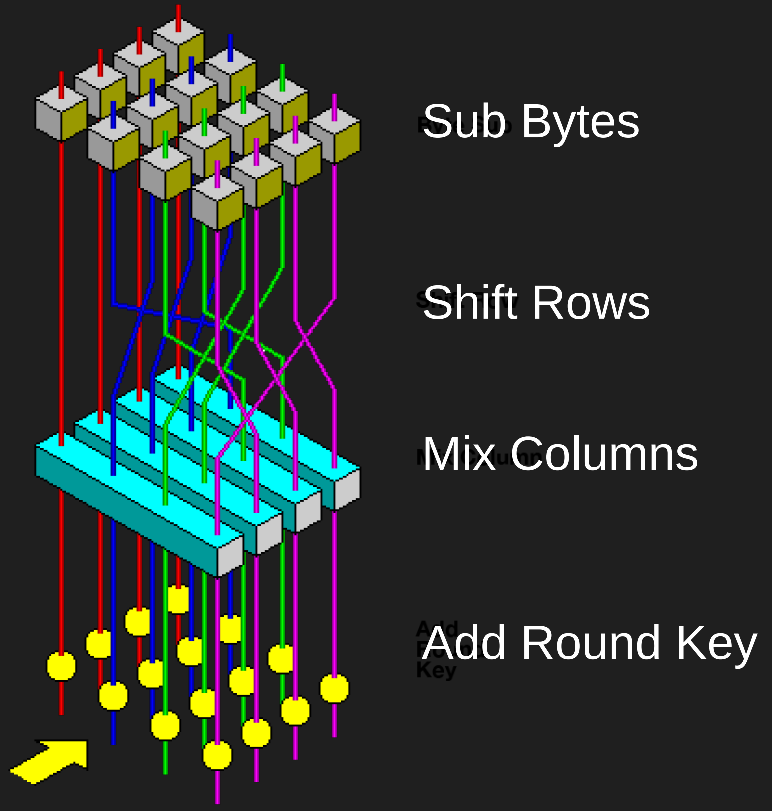
Sub Bytes (a nonlinear substitution using an S-box),

Shift Rows (a transposition step),

Mix Columns (mixing data within columns), and

Add Round Key (XOR with a round-specific key).

The first and last rounds are slightly modified, with the last round omitting the Mix Columns step.



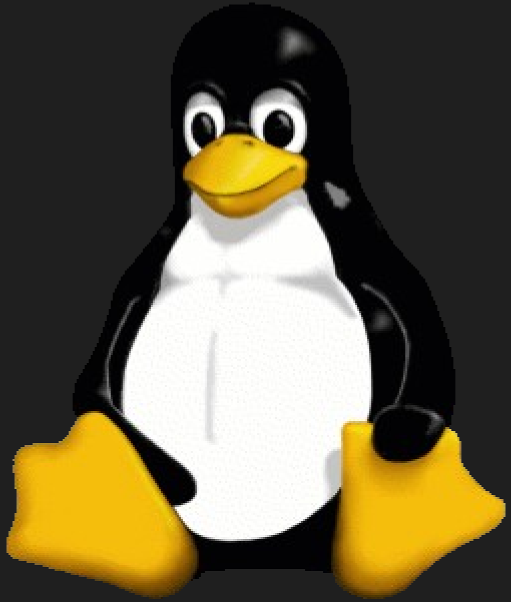
We Need to Consider Block Modes

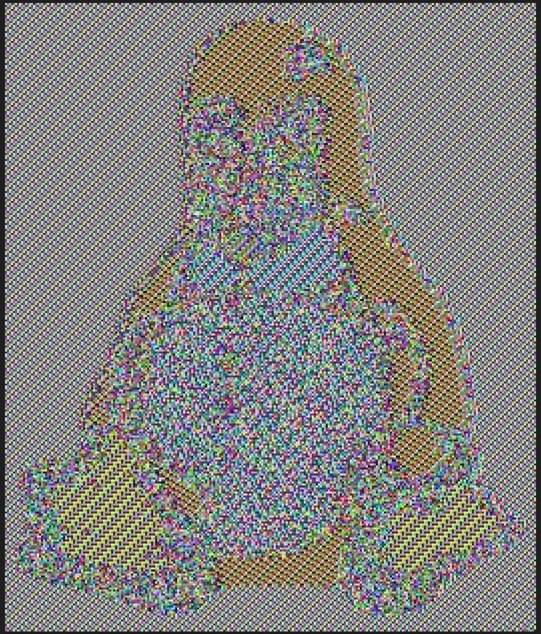
Common Block Modes

ECB – Electronic Code Book; fast and can be run in parallel across CPU threads. Input is cut up and the same key is applied to every block in the same way. This is fast, but can expose patterns which can be dangerous depending on your input data. This is the default if “AES” is specified as the algorithm in the encrypt() function as in our previous example. Not recommended.

“Tux” the Penguin in ECB Mode

“Tux” the Penguin in ECB Mode





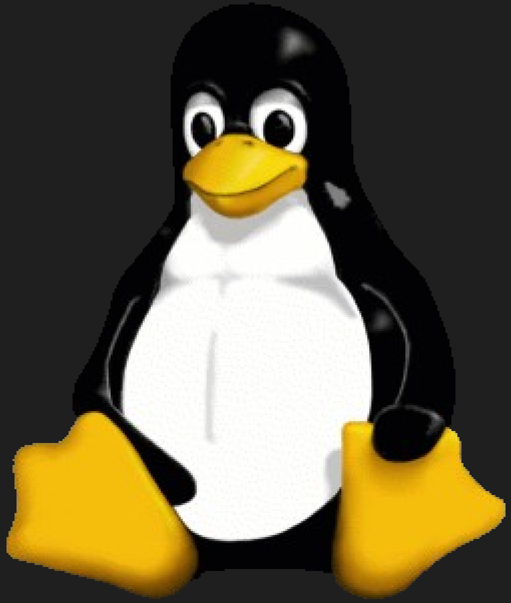
Common Block Modes

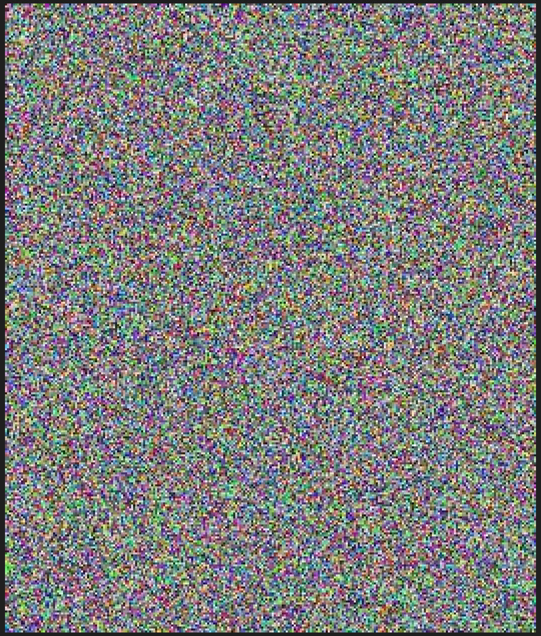
ECB – Electronic Code Book; fast and can be run in parallel across CPU threads. Input is cut up and the same key is applied to every block in the same way. This is fast, but can expose patterns which can be dangerous depending on your input data. This is the default if “AES” is specified as the algorithm in the encrypt() function as in our previous example. Not recommended.

CBC – Cipher Block Chaining; most secure; slow; must be run in a single thread. The output of each block is XORed with the previous key to create the key used on the next block. This results in each block being encrypted with a unique key which defeats patterns, but at the cost of slower processing. This is the new default after ColdFusion 2023 update 8 and ColdFusion 2021 update 14 when no algorithm is specified.

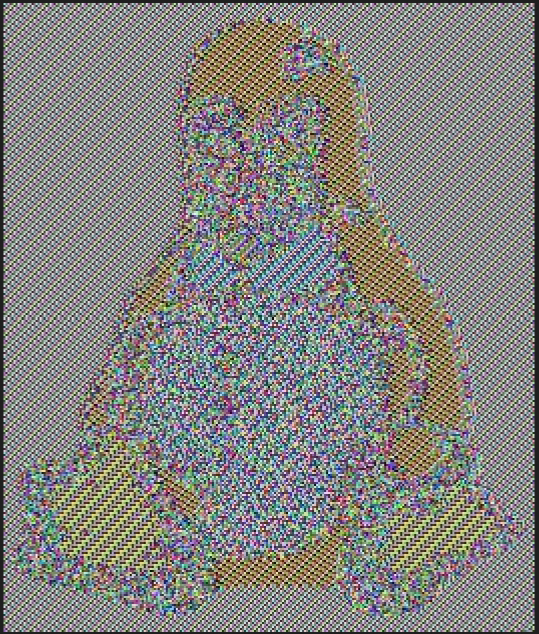
“Tux” the Penguin in CBC Mode

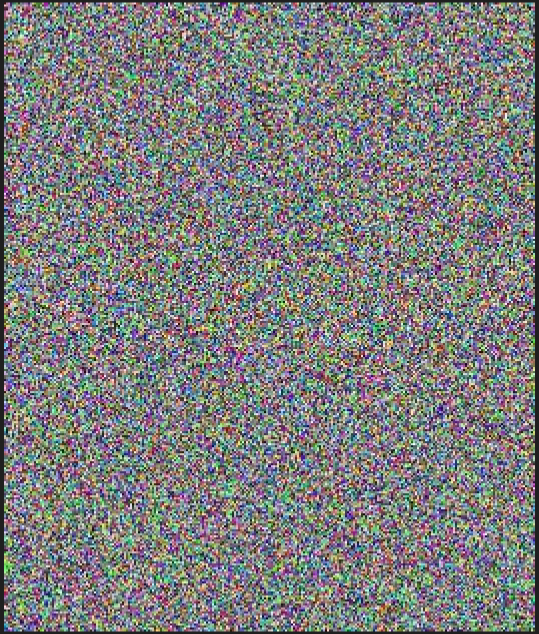
“Tux” the Penguin in CBC Mode





ECB vs CBC Modes





Common Block Modes

ECB – Electronic Code Book; fast and can be run in parallel across CPU threads. Input is cut up and the same key is applied to every block in the same way. This is fast, but can expose patterns which can be dangerous depending on your input data. This is the default if “AES” is specified as the algorithm in the encrypt() function as in our previous example. Not recommended.

CBC – Cipher Block Chaining; most secure; slow; must be run in a single thread. The output of each block is XORed with the previous key to create the key used on the next block. This results in each block being encrypted with a unique key which defeats patterns, but at the cost of slower processing. This is the new default after ColdFusion 2023 update 8 and ColdFusion 2021 update 14 when no algorithm is specified.

CTR / GCM – Counter Mode; reasonable compromise between speed and security. The original key is used to encrypt a nonce which gets incremented for each block to produce a keystream which is used to encrypt the plaintext. This allows it to be run in parallel because the key for each block is predictable and can be easily computed, but also secure because it prevents patterns by using a unique key for each block. GCM adds integrity checks to CTR.

Honorable Mentions

CFB – Cipher Feedback; similar to CTR in that it creates a keystream, but works on smaller chunks of data (e.g. 8 bits) rather than the full 128 bit block. It uses the previous block output as input for the next key, so has to run on a single thread like CBC mode. Does not require padding. Typical uses include working with stream-based data such as network traffic flows. I have never used it in web development, but it's supported by Java and ColdFusion, but we're not going to discuss it here.

OFB – Output Feedback; also similar to CTR in that is creates a keystream, but uses the previous block's key (like CBC) to create the next block key independent of the plain text or cipher text. It runs more slowly than CTR but isn't as secure as CBC. It has some special properties that make it good in situations where error propagation is undesirable, such as real-time communications over radio or satellite. Again, I've never had a use case for web development so we're going to ignore it for now.

encrypt() Examples with Block Mode

```
variables.key = generateSecretKey("AES"); // u8VVxZV9saMt7vUir5L+og== (128bit)
variables.plainText = "Secret String";

variables.ECB = encrypt(plainText, key, "AES/ECB/PKCS5Padding", "base64");
// wP+cd7Dk+I3RRSB56t1DOw== (128bits)
// Note also the same as "AES"

variables.CBC = encrypt(plainText, key, "AES/CBC/PKCS5Padding", "base64");
// MaDKKLHSn5DLX5+jXCTB/G35efFnyELKm2CeG6WL8yE= (256bits)
// Note this is the new default when no algorithm is specified

variables.CTR = encrypt(plainText, key, "AES/CTR/NoPadding", "base64");
// CUDOO7VuqptnfPIJnLU3YxXBJsi171ZJ/3Ox+vI= (232bits)
```

encrypt() Examples with Block Mode

```
variables.key = generateSecretKey("AES"); // u8VVxZV9saMt7vUir5L+og== (128bit)
variables.plainText = "Secret String";

variables.ECB = encrypt(plainText, key, "AES/ECB/PKCS5Padding", "base64");
// wP+cd7Dk+I3RRSB56t1DOw== (128bits)
// Note also the same as “AES”

variables.CBC = encrypt(plainText, key, "AES/CBC/PKCS5Padding", "base64");
// MaDKKLHSn5DLX5+jXCTB/G35efFnyELKm2CeG6WL8yE= (256bits)
// Note this is the new default when no algorithm is specified

variables.CTR = encrypt(plainText, key, "AES/CTR/NoPadding", "base64");
// CUDOO7VuqptnfPIJnLU3YxXBJsi171ZJ/3Ox+vI= (232bits)
```

Initialization Vectors

encrypt() Examples with Block Mode

```
variables.key = generateSecretKey("AES"); // u8VVxZV9saMt7vUir5L+og== (128bit)
variables.plainText = "Secret String";

variables.ECB = encrypt(plainText, key, "AES/ECB/PKCS5Padding", "base64");
// wP+cd7Dk+I3RRSB56t1DOw== (128bits)
// Note also the same as “AES”

variables.CBC = encrypt(plainText, key, "AES/CBC/PKCS5Padding", "base64");
// MaDKKLHSn5DLX5+jXCTB/G35efFnyELKm2CeG6WL8yE= (256bits)
// Note this is the new default when no algorithm is specified

variables.CTR = encrypt(plainText, key, "AES/CTR/NoPadding", "base64");
// CUDOO7VuqptnfPIJnLU3YxXBJsi171ZJ/3Ox+vI= (232bits)
```

encrypt() Examples with IV

```
variables.key = generateSecretKey("AES"); // u8VVxZV9saMt7vUir5L+og== (128bit)
variables.plainText = "Secret String"; // 104bits

variables.ECB = encrypt(plainText, key, "AES/ECB/PKCS5Padding", "base64");
// wP+cd7Dk+I3RRSB56t1DOw== (128bits)
// No IV required; ignored if specified.

variables.IV = binaryDecode(generateSecretKey("AES", 128), "base64");
// IV = binary decoded base64 JVR0KnGMdTkWW1NcsiI0IA==

variables.CBC = encrypt(plainText, key, "AES/CBC/PKCS5Padding", "base64", iv);
// j2Yzu6HyE6uNwGp8Z0hVAg== (128bits)

variables.CTR = encrypt(plainText, key, "AES/CTR/NoPadding", "base64", iv);
// d++/ZcnjA1lPQYp9Iw== (104bits)
```

Key Management

Key Sources

Simple string based keys (for CFMX_COMPAT):

```
createUUID(); // 6B65113E-8614-421E-AB87E8823E5818D4
```

Random, variable bit length keys for DES, DESEDE, AES, BLOWFISH, etc...

```
generateSecretKey("AES", 128); // u8VVxZV9saMt7vUir5L+og==
```

Derived from a user-provided password (all arguments required; base64 output):

```
generatePBKDFKey(
    "PBKDF2WithHmacSHA512", "password", salt, iterations, keysize
);
```

Externally provided

Prohibited Key Storage

Code – Do not include secrets in your code, especially encryption keys. Code gets shared, checked into source control, and can easily lead to your keys being accidentally leaked.

Default Configuration – If your application gets configured for specific clients or users, do not provide a “default” key that gets shared by multiple clients or users, as they will be unlikely to change these. In these cases force them to generate a new unique key as part of initial deployment.

Plain Text – Keys should never be stored in plain text, for example in configuration files, databases, etc.

With Ciphertext – It should go without saying that you should never store the key needed to decrypt ciphertext with the ciphertext, e.g. in the same database, unless it has itself been encrypted.

Ideal Key Storage

HSM – Hardware security module, such as the Thales Luna 7 series. These can be deployed as standalone appliances shared by multiple servers, PCIe cards installed into a server, or USB drives plugged into a server. These provide for key creation and lifecycle management, as well as encryption functions so keys never have to leave the device. Applications use an API or integration with Java for access to services. Expensive; can cost thousands of dollars to deploy.

Cloud-Based Vault – For cloud-hosted applications, distributed applications, or those on a budget, a cloud-based secrets vault can be a good option. Services such as AWS Secret Manager, Azure Key Vault, Google Secret Manager, Hashicorp Vault, Conjur, Keeper, Confidant, 1Password, BitWarden, etc. Typically paid based on usage in a given month.

Local – In a pinch, use local storage on the server such as the Windows Data Protection Application Programming Interface (DPAPI), a Java keystore file, the Windows Registry, or environment variables (these last two are not recommended, but options in a pinch).

Database Storage

If you must store symmetric keys in a database, e.g. alongside ciphertext, you must encrypt the key using encryption of the same strength or greater.

The key used to encrypt the key to your data is called a “Key Encrypting Key” (KEK) and should be stored in a secure manner away from the encrypted keys that it protects, such as in an HSM, Cloud Key Vault, etc.

If your KEK is weaker than the key you’re protecting, then the effective protection provided by your encryption is only as strong as the weakest link in that chain.

Credit Card Storage

Can’t Steal What You Don’t Store

“In general, no payment card data should **ever** be stored by a merchant unless it’s **necessary to meet the needs of the business**. Sensitive data on the magnetic stripe or chip must **never** be stored. Only the PAN, expiration date, service code, or cardholder name may be stored, and merchants must use technical precautions for safe storage.”

-PCI Security Standards Council

Source: https://listings.pcisecuritystandards.org/pdfs/pci_fs_data_storage.pdf

We Have a Business Need...

If you do need to store credit card numbers, for example to facilitate recurring billing for services, or user convenience for e-commerce, you need to look at Requirement 3 of the PCI-DSS requirements: “Protect Stored Account Data”.

This can include the Primary Account Number (PAN), Cardholder name, expiration date, and service code.

“Sensitive Authentication Data” including track data (magnetic stripe), card verification code, chip data, and PINs may never be stored after a transaction is completed, including in logs.

Consider storing the “truncated” card number instead (first six and last four): 432109…...4321 which does not require protection (unless paired with a hash)

Three Specific Requirements

3.5 – Primary account number (PAN) is secured wherever it is stored.

3.6 – Cryptographic keys used to protect stored account data are secured.

3.7 – Where cryptography is used to protect stored account data, key management processes and procedures covering all aspects of the key lifecycle are defined and implemented.

3.5.1 Allows “Strong Cryptography”

Requirement 3.5.1 says the “PAN is rendered unreadable anywhere it is stored by using [...] strong cryptography with associated key management processes and procedures.”

They define “Strong Cryptography” to mean:

“Cryptography is a method to protect data through a reversible encryption process, and is a foundational primitive used in many security protocols and services. Strong cryptography is based on **industry-tested and accepted algorithms** along with key lengths that provide a **minimum of 112-bits of effective key strength** and proper key-management practices.

Effective key strength can be shorter than the actual ‘bit’ length of the key, which can lead to algorithms with larger keys providing lesser protection than algorithms with smaller actual, but larger effective, key sizes. It is recommended that all new implementations use a minimum of 128-bits of effective key strength.”

3.6.1.2 Key Management

Requirement 3.6.1.2 defines the requirements for handling keys used for encryption:

“Secret and private keys used to protect stored account data are stored in one (or more) of the following forms at all times:

- Encrypted with a key-encrypting key that is at least as strong as the data-encrypting key, and that is stored separately from the data encrypting key.
- Within a secure cryptographic device (SCD), such as a hardware security module (HSM)...”

Code Example…

```
// Generate a 256 bit key for this card.
variables.key = generateSecretKey("AES", 256);

// Generate a 128 bit IV to use for this card.
variables.IV = binaryDecode(generateSecretKey("AES", 128), "base64");

// Encrypt the PAN using the new key and IV.
variables.panE = encrypt(form.pan, key, "AES/CBC/PKCS5Padding", "base64", iv);

// Encrypt the key with a KEK.
variables.kek = application.hsmService.getKey("cardKEK");
variables.keyE = encrypt(key, kek, "AES/CBC/PKCS5Padding", "base64", iv);

// Store the card with the order or customer account…
INSERT INTO orderpaymentcard (userID, cardName, pan, exp, cardKey, iv) VALUES (
    #userID#, '#form.cardName#', '#panE#', '#form.exp#', '#keyE#',
    '#binaryEncode(iv, "base64")#'
)
```

Key Takeaways

Hashing = Integrity, specifically for passwords

- If available, use the SCrypt (preferred) or BCrypt algorithms for password hashing and storage.
- Otherwise, use SHA-512 with 600,000 iterations or more and add a unique salt per password.
- Consider adding pepper before hashing for additional protection.
- Don’t use weak outdated defaults such as MD5, SHA1, etc.

Symmetrical Encryption = Data protection at rest

- Use AES unless you need compatibility with another application
- Use a 256 bit key unless a weaker key is needed for compatibility
- Use the correct block mode for your data
  - Avoid ECB if possible; it exposes patterns in your data (default if “AES” is specified with no block mode)
  - Use CBC mode for short data like PII or credit card numbers (most secure: “AES/CBC/PKCS5Padding”)
  - Use CTR or GCM for larger data like documents (balance of speed and security: “AES/CTR/NoPadding”)
- Generate and specify your own IV when possible for better compatibility

Key Management

- Never store encryption keys with encrypted data unless they themselves are strongly encrypted with a KEK.
- Never store secrets in code; accidental leaks from GitHub are more common than you think
- Keys should be stored securely using an HSM if possible, otherwise a Cloud-Based Vault.

Credit Card Numbers

- Don’t store unless you really need to. Use strong encryption; AES-CBC w/256 bit key. Use KEK for card keys.

Thanks!

Contact Info:

leviathan@darktech.org
www.darktech.org

www.linkedin.com/in/justinscott