Using Encryption in PHP

Ed Barnard, @ewbarnard

Image from my personal copy of "Cray in Chippewa Falls" by Lee Friedlander

Why Are We Here?

- Understand why it is so important — and difficult — to get it right
- Foundation: Two skills
  - Obtaining randomness
  - Encrypt and decrypt a string

Getting it Right

Never Roll Your Own Encryption! But…

- Third-party integrations may require you to do so
- You may not have newer, more modern PHP libraries/extensions in production
- Your legacy code base may already contain “roll your own encryption”
- Web services: Server and client may need access to same library

The OpenSSL Library/ Extension

- Using OpenSSL still means you are “rolling your own” cryptography!
- OpenSSL has only low-level tools:
  - Encrypt, Decrypt
  - HMAC
- It’s awfully easy to use these low-level functions **incorrectly**

Getting it Wrong

Ashley Madison data breach

From Wikipedia, the free encyclopedia

In July 2015, a group calling itself "The Impact Te[illegible]r data of Ashley Madison, a commercial website billed as enabling extra[illegible]e group copied personal information about the site's user base [illegible] release users' names and personally identifying information if Ashley M[illegible]mmediately shut down. On 18 and 20 August, the group leaked m[illegible]bytes of company data, including user details.

Because of the site'[illegible]eting users' personal information – including real names, home addresse[illegible] and credit card transaction records – many users feared being publicly [illegible]

Don't become famous this way!

Why are you Encrypting?

- Data storage and retrieval
  - Encrypt now for later retrieval/decryption
  - Easier because you control both ends
- Transmitting information
  - More difficult because you only control one end of transaction

The Problem

- You can’t know if you got it right until you decrypt the string
  - Success: Great. Done
  - Fail:
    You have no way to know what went wrong!

Encryption is Opaque by Design

- Did you call the decryption function correctly?
- Do you have the right secret key?
- Did you unpack/transform/transmit the secret key correctly?
- Did your encrypted string get truncated or mangled?
- Was the encryption wrong to begin with?
- You don’t even know where to ***start*** looking!

My Frustration: Web Service

- Server-side encryption responding correctly to requests from my development environment
- Production rejecting all client requests, claiming invalid encryption

The Cause

- Production *mbstring* out of date
- The development environment had been updated, with newer *mbstring*, when installing PHPUnit dependencies
- We don’t run PHPUnit in production, so did not do that dependency update in production
- I was using *mbstring* to chop apart raw binary secret keys
- Feature tests all ran perfectly, because same *mbstring* used round trip

Diagnosing the Problem

- Dumped out all intermediate encryption steps as hex and base64
- All looked fine in dev environment (no surprise given that dev environment was working)
- Dumping steps from production showed unexpected strings of zeroes
- Tracked this down to *mbstring* mangling the secret key

Lessons Learned

- Working with encryption is tough by design
- When something goes wrong, *no* information leaked as to what went wrong
- Why? We don’t want to guide our attacker in breaking our security
- Unfortunately here *you* are the attacker trying to figure out how to make it work
- Take a careful look at all dependencies (libraries, extensions, OS packages) across all environments

Obtaining Randomness

Randomness

- You need randomness because you need to keep secrets
- If a secret is easy to guess, it's not much of a secret
- The measure of *randomness* is ***entropy***
- Pick a number between **1** and **10**?
  - Daughter would always pick **7** because it is her lucky number
  - Not much uncertainty (**entropy**) in her "random" choice

Sixteen Million Model T Fords

- 2^{24} = 16,777,216 (~24 bits entropy)
- A few color choices: 2-3 bits entropy
- “Any customer can have a car painted any color that he wants so long as it is black.” — Henry Ford
- Zero entropy in the color choice

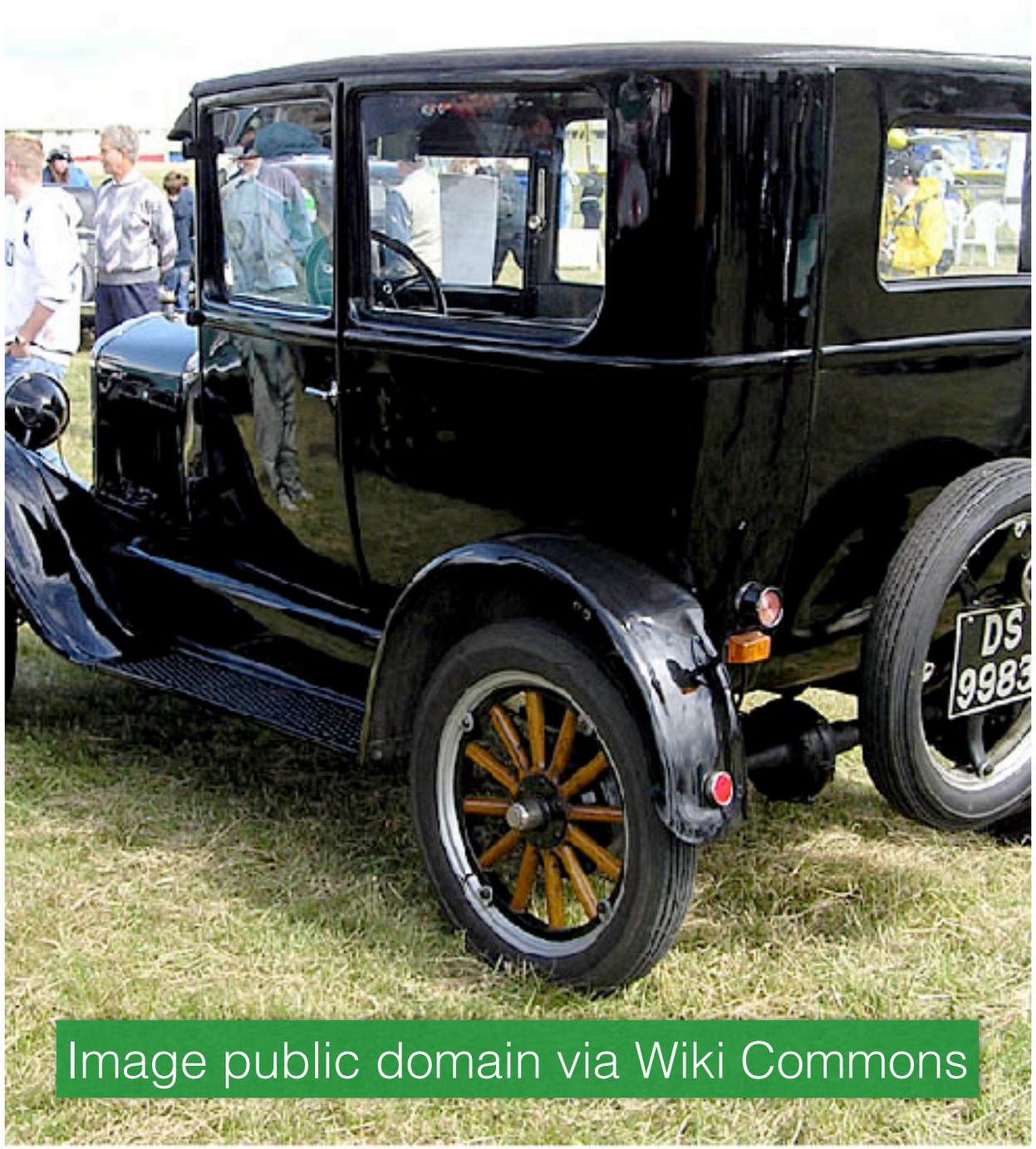

Image public domain via Wiki Commons

Less Randomness

- Pick a number between 1 and 10, but tend to pick even numbers
- English-language text
  - **A** more frequent than **Z**
  - **TH** (**th**is) more frequent letter combination than **TQ** (ou**tq**uote)
  - English-language text contains 1.5-2 bits of entropy per letter

A Random Model T

Using Randomness

- Real example: AES encryption with 256-bit keys in CBC mode
- Need 256-bit secret key
- Begin with password **123456** *(don't do this)*
- Be more secure! Use password **12345678**

Secret Key (Don’t Do This)

- Run **12345678** through SHA-256 and you have a 256-bit secret key:
  $secretKey = hash(‘sha256’, ‘12345678’, true);
- AES works just fine with your 256-bit derived **$secretKey**
- Anyone so stupid as to use **12345678** as their encryption password?
- All experienced attackers know the answer is **YES!**

What’s the Point? **Entropy**

- When encryption requires something that is
  - “random” or “unguessable”
  - “x” number of bits long
- That means you require ***that many bits of entropy***

12345678

- Given that **12345678** is on top-ten list of known passwords, you have *1-2 bits of entropy*, not the expected *256 bits of entropy*
- The sha256 function does *not* increase the entropy
- Your attacker only needs to guess the **12345678**
- Would an attacker check for something so obvious? Yes
- **You can stretch your 2 bits of entropy to a 256-bit value but it's still only 2 bits of entropy**

PHP Random Number Sources Fail

- Most PHP random-number sources have issues with predictability:
  - uniqid()
  - rand()
  - mt_rand()
  - openssl_random_pseudo_bytes()

Don’t use any of these functions as sources of randomness.

Not for cryptography, not for secret tokens, not for anything that should be unpredictable.

Linux: Use /dev/urandom

- On Linux systems, the best source of randomness comes from the Linux kernel as the `/dev/urandom` device
- This is `/dev/urandom` with a “u” not `/dev/random` without the “u” (both devices exist)

How to use /dev/urandom

- For PHP 5.x use https://github.com/paragonie/random_compat
- PHP 7.x has built-in functions (I'm not there yet)
- PHP *mcrypt* extension's function `mcrypt_create_iv()` can draw from `/dev/urandom`
- The *mcrypt* default changes between PHP versions; be sure tell it to use the right source of randomness

Conflicted Information

- Do not use *mcrypt* extension for cryptography. It has no active developer support even though it remains available for PHP 4.x, 5.x, 7.x
- However, *mcrypt*'s `mcrypt_create_iv()` may be your best-available source of randomness, because it gives you access to `/dev/urandom`

“Random” vs. “Urandom”

- `/dev/random` (without the “u”) is a “blocking” device unsuitable for web requests
- `/dev/urandom` is non-blocking, therefore suitable for web requests
- `/dev/random` will hang when it needs to obtain more randomness: Bad for web requests

Example

- `$secretKey = mcrypt_create_iv(32, MCRYPT_DEV_URANDOM);`
- First parameter is number of random bytes you want: 32 bytes is 256 bits
- This gives you 256 bits of entropy, which is what you want
- Use MCRYPT_DEV_URANDOM not MCRYPT_DEV_RANDOM
- Both sender and receiver need the above secret key; share in such a way there is no possibility of attacker obtaining/ observing $secretKey

Example: Session Token

- Goal: 128-bits of entropy, per *Cryptographic Engineering*
- Create an unguessable token
  - Upon seeing several tokens, no attacker can guess, predict, or generate future tokens
  - Use printable characters so token can be passed as part of web URL (query string parameter)

Roll Your Own

```
$random = mcrypt_create_iv(16,
MCRYPT_DEV_URANDOM);

$token = substr(base64_encode($random,
0, 22);

$token = str_replace(['/', '+'], ['-',
'_'], $token);
```

Use random_compat

```
    ],
    "require": {
        "php" : "~5.5|~7.0",
        "paragonie/random_compat": "^2.0"
    },
    "require-dev": {
        "phpunit/phpunit" : "~4.0||~5.0",
        "squizlabs/php_codesniffer": "~2.3"
    },
    "autoload": {
```

Use random_compat (2)

```php
namespace InboxDollars\GenerateToken;

class GenerateToken
{

    public function generateToken()
    {
        return substr(str_replace(['+', '/'], ['-', '_'],
            base64_encode(random_bytes(16))), 0, 22);
    }
}
```

Encrypting and Decrypting a String

Cryptographic Decisions

- Before making cryptographic decisions, find out whether you have libraries available to make the correct decisions for you
- Our example:
  - Web services
  - AES encryption in CBC mode with 256-bit key
  - Mobile app uses this method in talking to server

Cryptographic Integrity

- Encryption is pointless (false sense of security) unless you can guarantee the integrity of the transmission
- If an attacker modifies the encrypted message, you need to detect that fact
- HMAC: Hash-based Message Authentication Code
- HMAC requires *another* 256-bit secret key

Initialization Vector

- Our mode of encryption requires a random "starting point"
- If the same text is encrypted twice with the same secret key, the encrypted string needs to be different
- Starting point is called the *Initialization Vector* or *IV*
- *IV* is 16 bytes (128 bits)

Key Creation

- We need two secret keys
  - One for encryption
  - One for HMAC authentication
  - Each key needs to have 256-bits of entropy per our cryptographic design decision

Key Creation (2)

- We pull 64 bytes (512 bits) from our source of randomness
- Our source of randomness needs to be a *Cryptographically Secure Pseudo Random Number Generator* (**CSPRNG**)
- We are using a wrapper for `/dev/urandom` as our CSPRNG
- `$largeKey = base64_encode(mcrypt_create_iv(64, MCRYPT_DEV_URANDOM));`

Key Creation (3)

- Both sender and receiver need to securely retain copies of our `$largeKey`
- Remarkably tricky
- Do we send encryption key via un-encrypted email?
- Carrier pigeon? (public domain photo via wiki commons shows WWI soldier and carrier pigeon)

Key Derivation

We have a large key stored as a base64-encoded entity

- Decode the entity into raw data
- Take the left half, the first 32 bytes (256 bits)
- Take the right half, the second 32 bytes

```
$raw =
base64_decode($largeKey,
true);

$left = mb_substr($raw, 0, 32,
'8bit');



$right = mb_substr($raw, 32, 32,
'8bit');
```

Key Derivation (2)

We have a large key stored as a base64-encoded entity

- Create the encryption password as SHA-256 of the left half
- Create the authentication (HMAC) password as SHA-256 of the right half

```
$encryptionKey = hash('sha256', $left, true);

$authenticationKey = hash('sha256', $right, true);
```

What did we accomplish?

- SHA does not add any security to the encryption
- Neither the encryption key nor the HMAC authentication key are ever stored, anywhere
- In a mobile app, this approach might make it more difficult to extract the secret key
- We’re storing the 512-bit large key, and derive the two 256-bit keys as needed

Authentication (HMAC)

- There are long discussions around whether to *encrypt-then-authenticate* or *authenticate-then-encrypt*
- Let's just get to the code

Encrypt an Array

```
2   $data = array('a' => 1, 'b' => 2);
3   $message = json_encode($data);
4   $initializationVector = mcrypt_create_iv(16, MCRYPT_DEV_URANDOM);
5   $cipherText = openssl_encrypt($message, 'aes-256-cbc',
6       $encryptionKey, 1, $initializationVector);
7   $toCover = $initializationVector . $CipherText;
8   $hmac = hash_hmac('sha256', $toCover, $authenticationKey, true);
9   $result = base64_encode($hmac) . ':' .
10      base64_encode($initializationVector) . ':' .
11      base64_encode($cipherText);
```

Decrypt a String (1)

```
8   function doDecryptResult($result, $authenticationKey,
9       $encryptionKey) {
10      $result = (string)$result;
11      $results = explode(':', $result);
12      if (3 !== count($results)) {
13          return 'Invalid input string';
14      }
15      $hmac = base64_decode($results[0]);
16      $initializationVector = base64_decode($results[1]);
17      $cipherText = base64_decode($results[2]);
18      $toCover = $initializationVector . $cipherText;
19      $calculated = hash_hmac('sha256', $toCover,
20          $authenticationKey, true);
```

Decrypt a String (2)

```
21      if (!hash_equals($hmac, $calculated)) {
22          return 'Encrypted string not valid';
23      }
24      $message = openssl_decrypt($cipherText, 'aes-256-cbc',
25          $encryptionKey, 1, $initializationVector);
26      $unpacked = json_decode($message, true);
27      if (null === $unpacked) {
28          return 'Decrypted string not JSON';
29      }
30      return (array)$unpacked;
31  }
```

yan
@bcrypt

Following

one of the best slides ive ever made

RETWEETS 174 LIKES 497

3:53 PM - 30 Sep 2016

174 497

Summary

- Do what you need to do to get it right (Failure is *always* an option)
- Understand randomness, and how to get enough of it
- Understand encrypt/HMAC process
- Expect to do your homework. Failure can be *more* than merely embarrassing

Thank You

- Additional Reading: http://otscripts.com/using-encryption-in-php-madison-php-2016/
- Ed Barnard, InboxDollars.com
- ewbarnard@embarqmail.com
- Twitter @ewbarnard
- Slide Deck: (see joind.in)
- Rate this talk: https://joind.in/talk/49812