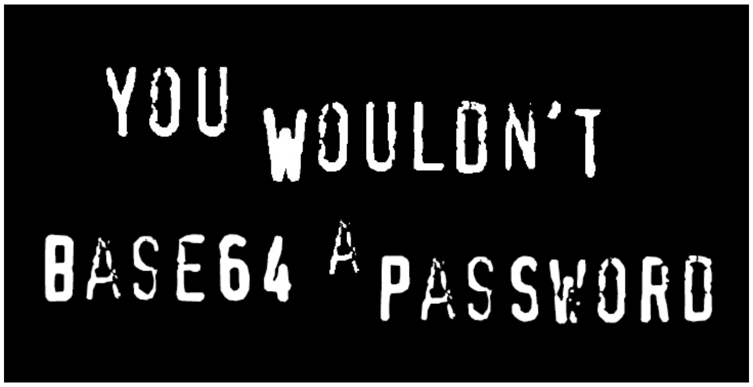
paragonie.com

You Wouldn't Base64 a Password - Cryptography Decoded

Paragon Initiative Enterprises

17-21 minutes



There's a ton of bad programming and security advice on the Internet. Some of the advice is bad because the author is misinformed, some because it emphasizes precision over clarity and most people wind up lost in the jargon.

If you feel that cryptography is a weird, complicated, and slightly intimidating subject for which your feelings might be best described as lukewarm (on a good day), we hope that by the time you finish reading this page, you will have a clear understanding of the terms and concepts people use when this topic comes up.

> Warning: The example snippets on this page are for illustrative purposes. Don't use them in your projects. If you want a real-world example to reference, check out the snippets in our Chief Development Officer's StackOverflow answer instead.

Let's start with a basic question: **What exactly is a cryptographic feature?** In the simplest terms we can muster: `Cryptographic features use math to secure an application`.

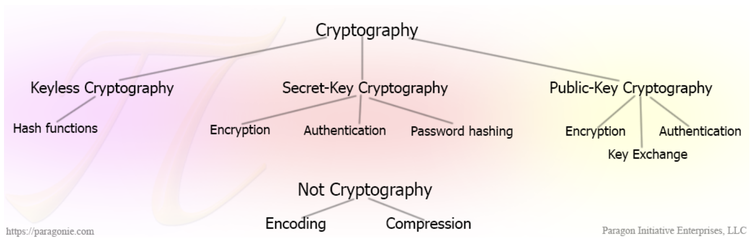
Digging a little deeper: there are a plethora of cryptography algorithms and they can generally be grouped together based on two criteria:

1. How much information must be supplied by the developer?
2. What is the intended goal?

- Confidentiality?
- Integrity?
- Authenticity?

- Non-repudiation? Deniability? (These two are opposites.)

Overview of Cryptography Concepts



- Keyless Cryptography (0 keys)
- Hash Functions
- Secret-Key Cryptography (1 key)
- Secret-Key Message Authentication
- Secret-Key Encryption
- Authenticated Secret-Key Encryption
- Public-Key Cryptography (2 keys)
- Shared Secret Key Agreement
- Digital Signatures

The First Rule of Cryptography: Don't Implement it Yourself

Developing cryptography features is best left to the experts. By all means, do feel free to tinker, but don't deploy your experiments in production or share them with other developers who might deploy them in production.

Instead, use a high-level cryptography library that experts have already vetted. Follow the link to read our PHP cryptography library recommendations.

Keyless Cryptography

The most simplest algorithm to consider is the **cryptographic hash function**, which accepts one input and returns a single deterministic fixed-size output.

```
hash("sha256", "");
//
e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855

hash("sha256", "The quick brown fox jumps over the
lazy dog");
//
d7a8fbb307d7809469ca9abcb0082e4f8d5651e46d3cdb762d02d0bf37c9e592
```

When using a well-designed cryptographic hash function, such as **BLAKE2** or SHA256, any change you make to the message will result in a drastically different hash output.

```
hash("sha256", "The quick brown fox jumps over the
lazy cog");
```

Simple hash functions are fast and deterministic; if you have any arbitrary message, you can calculate the hash output for that particular message. By themselves, they are mostly useful for error checking or as a building block for other cryptographic primitives, which most developers will not need to develop.

Cryptographic hash functions are one-way data transformations. Although you can easily calculate the hash output (often referred to as a message digest) for any arbitrary message, you cannot easily go from the hash output to the original message.

Some hash functions (such as MD5) have weaker security guarantees and smaller output sizes. As a result, it's almost trivial to calculate two different messages that will produce the same MD5 hash.

Secret Key Cryptography

Most cryptography algorithms aren't as simple as hash functions. As a consequence, they are a lot more useful and can provide security guarantees beyond, "Yes, this output can be reproduced from this input."

Consequently, they typically require two pieces of input: The message and a **secret key**. A secret key should be a unique string of random bytes that both the sender and intended recipient should know, and nobody else.

Keyed Hash Functions - Message Authentication

A keyed hash function, such as HMAC, is a special implementation of a hash function that accepts a message and a secret key and produces a Message Authentication Code (MAC).

```
hash_hmac("sha256", "The quick brown fox jumps
over the lazy dog", "secret key");
```

```
hash_hmac("sha256", "The quick brown fox jumps
over the lazy cog", "secret key");
```

```
hash_hmac("sha256", "The quick brown fox jumps
over the lazy dog", "secret kez");
```

```
hash_hmac("sha256", "The quick brown fox jumps
over the lazy cog", "secret kez");
```

Keyed hash functions are more useful than hash functions; only someone with the secret key can calculate a MAC for a given message. Therefore, if you transmit a message and a MAC for a given message, and never transmit the secret key, you can be reasonably sure that the message is authentic.

Secret Key Encryption

Warning: Encryption without message authentication is vulnerable to chosen ciphertext attacks. Please read our whitepaper on Secure Data Encryption in PHP.

Formally, encryption is the reversible process of transforming of a message (called the `plaintext`) and a **secret key** into a seemingly random string of bytes (called the `ciphertext`). i.e. `encrypt($message, $key)` should return a unique string of random bytes for a given pair of `$message` and `$key`.

Unfortunately, simple secret-key encryption (also known as ECB mode) is not secure. If you encrypt the same (16-byte, for the popular AES encryption algorithm) block within a message with the same key in ECB mode, the ciphertext will repeat.

Modern secret-key encryption, therefore, actually accepts more than two pieces of information. Beyond the `plaintext` message and a **secret key**, they also require a unique *Initialization Vector* (IV, for CBC mode) or *nonce* (number to be used once, for CTR mode). The difference between a nonce and IV is subtle.

None of the code on this page is secure; neither are any of the encryption keys.

```
bin2hex(
    openssl_encrypt(

        "The quick brown fox jumps over the lazy
dog",

        'aes-128-ctr',

        "\x01\x02\x03\x04" . "\x05\x06\x07\x08" .
"\x09\x0a\x0b\x0c" . "\x0d\x0e\x0f\x10",

        OPENSSL_RAW_DATA,

        str_repeat("\0", 16)
    )
);



openssl_decrypt(

    hex2bin(

"8f99e1315fcc7875325149dda085c504fc157e39c0b7f31c6c0b333136a7a8877c4971a
    ),

    'aes-128-ctr',

    "\x01\x02\x03\x04" . "\x05\x06\x07\x08" .
"\x09\x0a\x0b\x0c" . "\x0d\x0e\x0f\x10",

    OPENSSL_RAW_DATA,

    str_repeat("\0", 16)
);
```

A more in-depth and less illustrative example (which properly generates IVs) is available here.

For a closer examination at symmetric-key encryption with OpenSSL, read our white paper.

Decryption is only successful if the same IV/nonce and secret key are used. However, only the key must be kept secret; the IV and nonce can even be broadcast with your encrypted message.

Authenticated Secret-Key Encryption

If you recall from our earlier blog post, *Using Encryption and Authentication Correctly*, secret-key encryption itself is vulnerable to tampering unless you combine it with authentication.

The only strategies proven to be secure are to use an AEAD mode or to always **encrypt first then authenticate the encrypted data** with a MAC.

If you are following an Encrypt-Then-MAC construction, you want to use two separate secret keys: One for the encryption, the other for the MAC. In other words, apply the previous two sections together:

```
$nonce = random_bytes(16);
$ciphertext = openssl_encrypt(

    "The quick brown fox jumps over the lazy dog",
    'aes-128-ctr',

    "\x01\x02\x03\x04" . "\x05\x06\x07\x08" .
"\x09\x0a\x0b\x0c" . "\x0d\x0e\x0f\x10",

    OPENSSL_RAW_DATA + OPENSSL_ZERO_PADDING,

    $nonce
);

$mac = hash_hmac("sha256", $nonce.$ciphertext,
"\xff\xfe\xfd\xfc" . "\xfb\xfa\xf9\xf8" .
"\xf7\xf6\xf5\xf4" . "\xf3\xf2\xf1\xf0", true);
echo bin2hex($nonce.$ciphertext.$mac);
```

It is important to exercise caution when combining cryptographic features. Our basic protocol as written above has no redundant features:

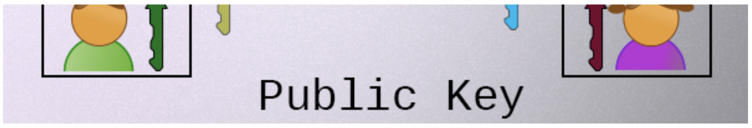
- Secret key encryption provides **confidentiality** such that it can only be read with the correct secret key.
- Keyed hash functions provide **authentication** (and consequently, *message integrity*) such that anyone possessing the correct secret key can recalculate the same MAC.
- A random IV/nonce is used to make each encrypted message unique, *even if the unencrypted message is the same*.

It should go without saying, but double-encrypting or double-authenticating when you need authenticated encryption would just be silly.

Public Key Cryptography

Public key cryptography is challenging for nontechnical people to understand, and even more challenging for technical people to explain correctly without burying the reader in mathematics or missing critical points. The end result is usually a lot of confusion and occasionally a false sense of understanding. (A fauxreka moment, if you will.)

Secret Key
DJ2o0SJl...
...+29A0Kmn/==



Here's all you need to know right now: Unlike secret key encryption, which involves a single secret key that is held by both parties, in public key cryptography, each participant has **two keys**:

- Each participant has a **private key**, which they never share.
- Each participant also has a **public key**, which is mathematically related to their private key, which they share with everyone.

It is unfortunate that the "key" terminology from secret key cryptography stuck when public key cryptography was discovered, as there aren't very many physical systems that are intuitively similar to what's going on here. Some people have attempted to explain public key cryptography using colors or detailed explanations. If you're interested in the intimate details, we recommend both of the links in the previous sentence.

For everyone else, if you can accept these premises, understanding the rest isn't hard:

- To use public key cryptography, you generate a key-pair and share the public key, but keep the private key to yourself. (In most cases, every participant does this.)
- There is only one private key for any given public key.
- Both of the keys in a given key-pair are related to each other, mathematically.
- Given a public key, it is almost impossible to figure out what the private key is.
- Given a private key, you can near-instantly calculate the related public key.

Got it? Let's build something with this understanding.

Shared Secret Key Agreement

Let's say you want to talk to a friend over the Internet using secret key cryptography (which is much faster than public key cryptography), but you don't want anyone else to read it. You and her haven't already agreed upon a secret key. How do you do it?

Glossing over the finer details (the color video above explains it fairly well), this is what you do:

1. You send her your public key (yellow).
2. She sends you her public key (light blue).
3. Combine your private key (green) and her public key (blue) to form a shared secret key.

4. She will combine her private key (red) with your public key (yellow) to form **the same exact shared key**.

   How? Modular arithmetic (classic Diffie Hellman) or multiplication along elliptic curves over finite fields (modern Elliptic Curve Diffie Hellman).

Digital Signatures

Digital signature algorithms, such as EdDSA (Edwards-curve Digital Signature Algorithm), are one of the most useful innovations to result from public key cryptography.

A **digital signature** is calculated from a **message** and a **private key**. Earlier algorithms (such as ECDSA) also required you to generate a unique random nonce for each message, but this was proven to be error-prone in the real world.

Anyone else with a copy of your **public key** can verify that a particular message was signed by your private key. Unlike keyed hash functions, this verification takes place without requiring you to reveal your private key.

Password Storage

Quick answer: Just use bcrypt. For PHP developers, this means `password_hash()` and `password_verify()` rather than `crypt()`.

Many developers think passwords should be *encrypted*, but this is false. **Passwords should be *hashed***, not encrypted. Furthermore, don't confuse password hashing algorithms with simple cryptographic hash functions. They're not the same thing:

Cryptographic Hashes	Password Hashes
• Fast • Only one input: The message	• Intentionally slow • At least three inputs: 1. The password 2. A per-user salt 3. A cost factor (how expensive to make the computation)

Unlike cryptographic hashes, password hashes require more than one input parameter. But unlike encryption algorithms, password hashes are one-way deterministic trap door calculations. Also unlike secret-key encryption, the salt does not need to remain secret; it merely needs to be unique per user. The purpose of a unique salt per user is to thwart pre-computation and to make brute-force guessing passwords from a list of hashes more expensive.

Can I encrypt my (bcrypt) password hashes?

Yes. If you run your web application and your database on separate hardware servers, this actually provides a substantial defense in depth. That's the reasoning behind our password_lock library.

File Verification

Digital signatures can prove authenticity, cryptographic hash functions can not.

There is a nontrivial portion of technical users that will, upon downloading an executable from a website, recalculate the MD5 or SHA1 hash of the file and compare it to one displayed on the web page they downloaded the file from. If it matches, they will execute the file, fully trusting its contents to be genuine.

If both the file and the hash value are stored on the same server, this is a completely ludicrous waste of time: **Any attacker who can alter your download can replace the hashes on the web page too.** (If the file and hash are on separate servers, the situation is a little different, but the improvement is not significant enough to warrant eschewing a better solution.)

After all, as we said above, hash functions like MD5 and SHA1 produce a deterministic fixed-size output for a given input. There are no secrets involved. When a solution does not increase security but makes people feel more secure, we call it security theater.

Cryptographic hash functions are security theater in this situation. You want **digital signatures** instead.

To improve security, instead of posting MD5/SHA1 hashes, the software vendor can instead sign their package with their EdDSA private key and share their EdDSA public key far and wide. When you download the file, you should also download the signature and, using the verified public key, check that it is authentic.

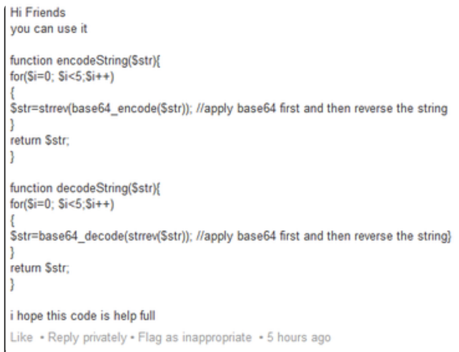
For example: Minisign.

A keyed hash function won't work here either, as you would need to distribute the secret key in order for anyone to be able to verify the signature. If they have the secret key, they can forge their own signatures for maliciously altered message (in this case, executable file).

Digital signatures are the best way to achieve assurance about the authenticity of a download. MD5/SHA1 hashes are almost always useless here.

Encoding and Compression Aren't Cryptographic

A common beginner's mistake is to use an encoding function, such as `base64_encode()`, to attempt to obfuscate information. Consider the following code, which was offered in a LinkedIn discussion about how to properly store passwords in a PHP web application:



Hi Friends
you can use it

```
function encodeString($str){
for($i=0; $i<5;$i++)
{
$str=strrev(base64_encode($str)); //apply base64 first and then reverse the string
}
return $str;
}

function decodeString($str){
for($i=0; $i<5;$i++)
{
$str=base64_decode(strrev($str)); //apply base64 first and then reverse the string}
}
return $str;
}
```

i hope this code is help full

Like • Reply privately • Flag as inappropriate • 5 hours ago

This may very well be the worst password storage function ever written.

A lot of developers will either encode or compress information and assume their solution provides the same level of security as actual cryptographic features simply because the output is not human readable. It doesn't.

Encoding and compression algorithms are both **reversible, keyless transformations of information**. Encoding specifies how information should be represented in human-readable text. Compression attempts to reduce an input to as little space as possible. Both are useful, but they are not cryptographic features.

- **Cryptographic hash algorithms** (e.g. SHA256) are deterministic one-way algorithms that require `zero` keys.
- **Keyed hashing algorithms** (e.g. HMAC) are used for authentication in secret-key cryptography; requires `one` key.
- **Secret-key encryption algorithms** (e.g. AES-CTR) are used to transform messages so only someone possessing the secret key can reverse; requires one `key`.
- **Shared secret agreement algorithms** (e.g. ECDH) are used to negotiate a shared secret key while only requiring the public transmission of both party's public keys. Requires `four` keys (two pairs of private/public) to generate a fifth.
- **Digital signature algorithms** (e.g. Ed25519) are used to sign messages (with one's private key) that anyone possessing the corresponding public key can validate. Requires `two` keys.
- **Password hashing algorithms** (e.g. bcrypt) are slow hashing algorithms designed specifically for being difficult to efficiently attack with a brute force search. Requires `one` secret input and a per-user salt.
- **Encoding algorithms** (e.g. Base64) are not cryptographic.
- **Compression algorithms** (e.g. gzip) are not cryptographic.

Keep in Mind

- **Don't encrypt passwords.** Instead, hash them with a password hashing algorithm. (You may encrypt the hashes.) Hash functions like MD5, SHA1, and SHA256 are not encryption. Anyone who uses the phrase "password encryption" probably needs to read this entire page carefully, because they are deeply mistaken.
- Secret-key encryption without message authentication is insecure (it's vulnerable to chosen ciphertext attacks).
- For downloads: digital signatures prove authenticity, hashes do not. You want a Minsign or GPG signature, not an MD5 hash.

We hope that this post serves as a good introduction to cryptography concepts. Our team publishes new posts about cryptography, application security, and web development in PHP anywhere from 2 to 5 times per month (usually on Friday). We also offer code review and technology consulting services.