

使用 Cryptomator 实现云同步端到端加密

「简介」

Cryptomator是为解决在将文件保存到云存储时可能出现的隐私问题而设计的，是一款开源的，安全的，没有后门的文件加密软件，可以轻松的对文件进行加解密，并在锁定文件目录后，自动退出文件目录，不留“痕迹”。

Cryptomator可以有效缓解云提供商或第三方未经许可访问云中存储的数据的风险。只有知道保险库密码的人才读取库中的文件或在不被察觉的情况下更改文件内容。这对文件内容和文件名都是适用的。

特点

- 与 Dropbox、Google Drive、OneDrive、百度网盘、阿里云盘、坚果云和任何其他与本地目录同步的云存储服务一起使用
- 客户端：没有账户，没有与任何在线服务共享的数据
- 完全透明：只需在虚拟驱动器上工作，就像是 usb 闪存一样
- 256 位密钥长度的 AES 加密
- 文件名被加密
- 在你的 Dropbox 中使用任意数量的储藏室，每个储藏室都有单独的密码
- 4000 次提交数据的安全性

注意事项

为了允许与云的有效同步，Cryptomator不加密一些metadata。这些元信息包括：

- 文件和文件夹的访问、修改和创建时间戳，
- 保险库和文件夹中的文件和文件夹数量，
- 存储文件的大小。

如果在处理加密文件时创建了备份副本，Cryptomator本身无法检测这些备份文件。Cryptomator只关注对库中文件的加密和解密，对于文件的备份不提供直接的监测或管理。

Cryptomator的密码是存在本地的。Cryptomator使用在创建保险库（vault）时设置的密码来加密和解密文件。

「加密架构」

Virtual Filesystem

Cryptomator提供了一个虚拟驱动器，可以像在PC上使用任何磁盘驱动器一样，添加、编辑、删除文件，而在Cryptomator每次都是加密这个驱动器内的文件。

目前来说Cryptomator试用 **FUSE** 来实现用户态文件系统，因此选择使用WinFSP（在Windows上）、macFUSE（在macOS上）和FUSE（在Linux上）作为他们的首选前端。如果都不可用的话，就会回退到WebDAV协议。

Vault Configuration

每个Vault必须在其根目录中有一个名为 `vault.cryptomator` 的Vault配置文件。它是一个 `JWT`，包含有关Vault的基本信息和要使用的密钥的规范。`JWT` 使用512位的原始主密钥进行签名。

以下是一个编码的Vault配置文件的示例：

```
eyJraWQiOiJtYXN0ZXJrZXlmaWxlOm1hc3RlcmtleS5jcnlwdG9tYXRvciIsInR5cCI6IkpXVCIsImFsZyI6IkhTMjU2In0.eyJmb3JtYXQiOjgsInNob3J0ZW5pbmdUaHJlc2hvbGQiOjIyMCwianRpIjoiyY2U5NzZmN2EtN2I5M0Y2MwLWI0YzEtYzcyOTZlY2Y1IiwiaWF0Ij0iJTSVZfQ1RSTUFDIiwiaWF0Ij0iJTI0In0.IJlu4dHb3fqB2fAk9Lf8G8zyEXc70LB-5m9aNx0EXIQ
```

解码的头部(header)

```
{
  "kid": "masterkeyfile:masterkey.cryptomator", /* 获取密钥的位置的URI */
  "typ": "JWT",
  "alg": "HS256" /* 当前实现还支持HS384和HS512 */
}
```

解码的payload:

```
{
  "format": 8, /* vault format for checking software compatibility 兼容性*/
  "shorteningThreshold": 220, /* how many characters in ciphertext filenames
before shortening */
  "jti": "ce976f7a-7b92-4cc0-b4c1-c74a6aa17cf5", /* random UUID to uniquely
identify the vault */
  "cipherCombo": "SIV_GCM" /* 块密码的操作模式。其他可能的值是"SIV_CTRMAC" */
}
```

打开Vault时，必须按照以下步骤进行：

1. 解码 `vault.cryptomator`，无需验证。
2. 读取 `kid` 头部，并根据其值从指定位置检索主密钥。
3. 使用主密钥验证 `JWT` 签名。
4. 确保支持 `format` 和 `cipherCombo` 字段。

Masterkey Derivation 密钥派生

每个 `vault` 都有自己的256位加密主密钥和MAC主密钥，分别用于加密文件特定密钥和文件身份验证。

这些密钥是由 `CSPRNG` 生成的随机序列，使用的是`SecureRandom`的SHA1PRNG模式，种子是由 `SecureRandom.getInstanceStrong()` 生成的440位seed。

这两个密钥都使用RFC 3394密钥封装，使用从用户的密码中派生的KEK（使用`scrypt`）进行加密。

```
// 生成一个长度为32字节的随机数序列，作为加密主密钥
encryptionMasterKey := createRandomBytes(32)

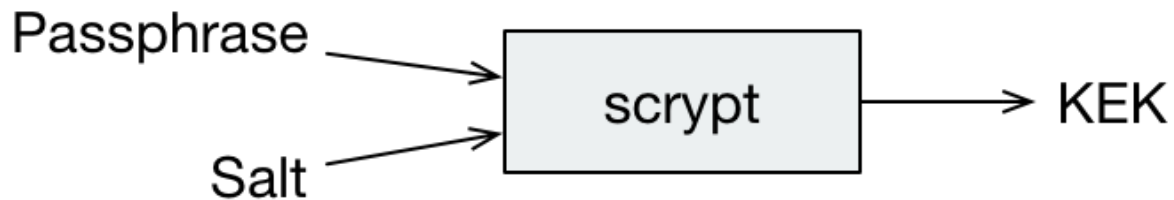
// 生成一个长度为32字节的随机数序列，作为MAC主密钥
macMasterKey := createRandomBytes(32)

// 使用scrypt函数从用户提供的密码、盐、成本参数和块大小派生出密钥加密器（KEK）
```

```
kek := scrypt(password, scryptSalt, scryptCostParam, scryptBlockSize)

// 使用AES密钥封装 (Wrap) 加密主密钥，以便安全地存储
wrappedEncryptionMasterKey := aesKeyWrap(encryptionMasterKey, kek)

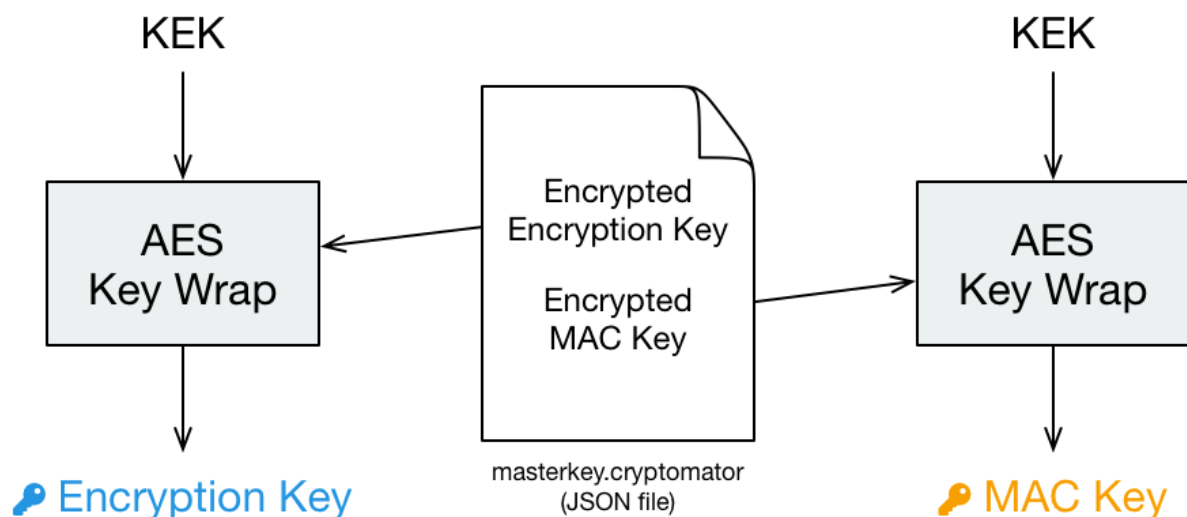
// 使用AES密钥封装 (Wrap) MAC主密钥，以便安全地存储
wrappedMacMasterKey := aesKeyWrap(macMasterKey, kek)
```



然后，封装的密钥和派生KEK所需的参数以整数或Base64编码的字符串形式存储在名为 `masterkey.cryptomator` 的JSON文件中，该文件位于Vault的根目录中。

```
{
  "version": 999, /* deprecated, vault format is now specified in the vault
configuration */
  "scryptSalt": "QGk...jY=",
  "scryptCostParam": 16384,
  "scryptBlockSize": 8,
  "primaryMasterKey": "QDi...Q=", /* wrappedEncryptionMasterKey */
  "hmacMasterKey": "L83...Q=", /* wrappedMacMasterKey */
  "versionMac": "3/U...9Q=" /* 防止未检测到的降级攻击的HMAC-256 */
}
```

解锁保管库时，KEK 用于解开（即解密）存储的主密钥。



```
// 伪代码表示解锁Vault过程

// 从`masterkey.cryptomator`文件中读取存储的信息
scryptSalt = readFromFile("masterkey.cryptomator", "scryptSalt")
scryptCostParam = readFromFile("masterkey.cryptomator", "scryptCostParam")
scryptBlockSize = readFromFile("masterkey.cryptomator", "scryptBlockSize")
```

```

wrappedEncryptionMasterKey = readFromFile("masterkey.cryptomator",
"primaryMasterKey")
wrappedMacMasterKey = readFromFile("masterkey.cryptomator", "hmacMasterKey")

// 用户提供的密码
password = getUserPassword()

// 使用scrypt函数从用户提供的密码、盐、成本参数和块大小中派生密钥加密器 (KEK)
kek = scrypt(password, scryptSalt, scryptCostParam, scryptBlockSize)

// 使用AES密钥解封装 (Unwrap) 加密主密钥, 得到原始的加密主密钥
encryptionMasterKey = aesKeyUnwrap(wrappedEncryptionMasterKey, kek)

// 使用AES密钥解封装 (Unwrap) MAC主密钥, 得到原始的MAC主密钥
macMasterKey = aesKeyUnwrap(wrappedMacMasterKey, kek)

// 现在, encryptionMasterKey和macMasterKey可以用于后续的加密和身份验证操作

```

File Header Encryption

目的: 存储文件内容加密所需的元数据。

大小: 包含68字节。

组成部分:

- 12字节用于头有效负载加密的nonce。
- 40字节AES-GCM加密有效负载, 包含了必要的元数据, 如nonce和文件内容密钥
 - 8字节填充为1, 供将来使用 (先前用于文件大小)。
 - 32字节文件内容密钥。
- 16字节加密有效负载的标签, 用于验证文件头的完整性和真实性。

```

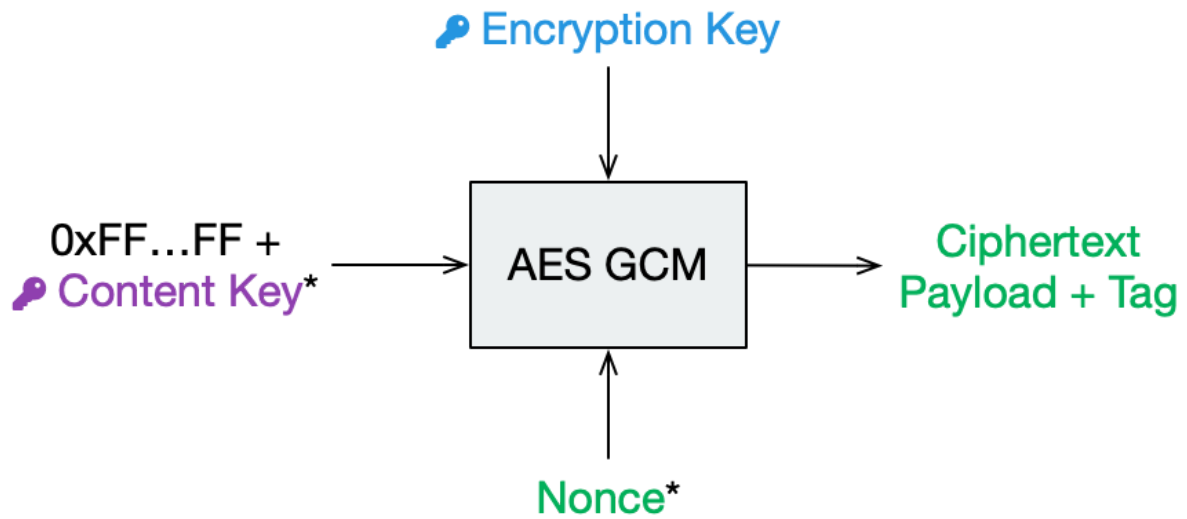
// 生成一个随机的12字节nonce, 用于头有效负载的加密
headerNonce := createRandomBytes(12)

// 生成一个随机的32字节文件内容密钥
contentKey := createRandomBytes(32)

// 创建明文有效负载, 将32字节文件内容密钥附加到0xFFFFFFFFFFFFFFFF
cleartextPayload := 0xFFFFFFFFFFFFFFFF . contentKey

// 使用AES-GCM算法对明文有效负载进行加密, 得到密文有效负载和标签
ciphertextPayload, tag := aesGcm(cleartextPayload, encryptionMasterKey,
headerNonce)

```



File Content Encryption

目的： 加密实际的文件内容。

明文会被分割为多个块（每个块最多32 KiB + 28字节），每个块包括：

- 12字节nonce。
- 最多32 KiB使用文件内容密钥的AES-GCM加密的有效负载。
- 16字节标签，通过GCM计算，带有特定的附加认证数据（AAD）：
 - 块编号作为32位大端整数（防止未检测到的重新排序）。
 - 文件头nonce（将块与文件头绑定）。

加密块连接在一起，保留明文块的顺序。最后一个块的有效负载可能小于32 KiB。

```
// 将明文数据分割成最大为32 KiB + 28字节的块
cleartextChunks[] := split(cleartext, 32KiB)

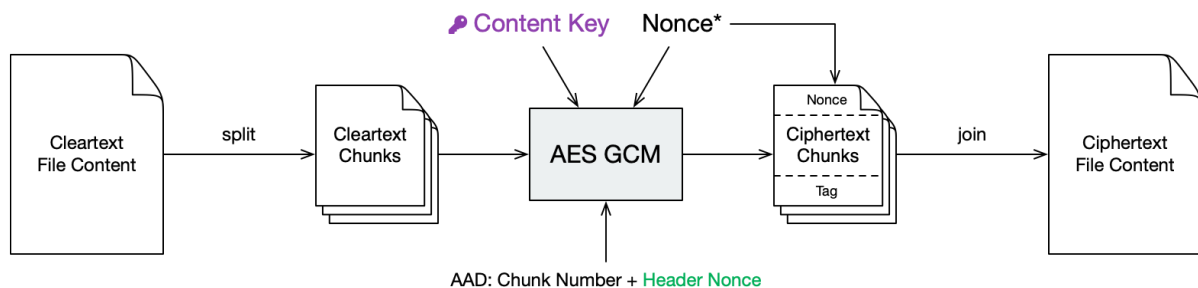
// 对每个明文块进行处理
for (int i = 0; i < length(cleartextChunks); i++) {
    // 生成一个随机的12字节nonce，用于块的加密
    chunkNonce := createRandomBytes(12)

    // 准备附加认证数据（AAD）：使用大端序表示的块编号和文件头nonce
    aad := [bigEndian(i), headerNonce]

    // 使用AES-GCM算法对明文块进行加密，得到密文有效负载和标签
    ciphertextPayload, tag := aesGcm(cleartextChunks[i], contentKey,
    chunkNonce, aad)

    // 将生成的密文块存储，包括nonce、密文有效负载和标签
    ciphertextChunks[i] := chunkNonce . ciphertextPayload . tag
}

// 将所有密文块连接在一起，形成整个文件内容的密文
ciphertextFileContent := join(ciphertextChunks[])
```



Directory IDs

每个目录都有一个唯一的ID，在文件名加密时需要使用该ID。出于历史原因，目录ID是一个字符串，尽管任何字节序列都能完成任务。

根目录的目录ID为空字符串。对于所有其他目录，它是一个最多包含36个ASCII字符的随机序列（建议使用随机UUID）。

```
dirId := createUuid()
```

遍历目录时，会分四个步骤处理给定子目录的目录 ID，以确定Vault内的存储路径：

1. 使用AES-SIV加密目录ID以加密目录层次结构。
2. 创建加密目录ID的SHA1哈希，以获得统一长度。
3. 使用Base32对哈希进行编码，以获取可打印字符的字符串。
4. 用Base32编码的哈希构建目录路径。

```

dirIdHash := base32(sha1(aesSiv(dirId, null, encryptionMasterKey,
macMasterKey)))
dirPath := vaultRoot + '/d/' + substr(dirIdHash, 0, 2) + '/' + substr(dirIdHash,
2, 30)

```

无论明文路径的层次结构如何，密文目录始终以扁平结构存储。因此，所有目录实际上都是兄弟姐妹（或者准确说，是表兄弟姐妹）。

Filename Encryption

文件的明文名称是在Normalization Form C中使用UTF-8进行编码。

Cryptomator使用AES-SIV对名称进行加密。父文件夹的目录ID被传递为 **关联数据**。这可以防止文件在目录之间被未检测到地移动。

在具体实现中，父文件夹的目录ID在AES-SIV加密算法中作为关联数据传递，以确保每个文件名都与其所属目录的特定信息相关联。这样做可以增加加密的安全性，防止攻击者试图通过将文件从一个目录移动到另一个目录来规遍加密。

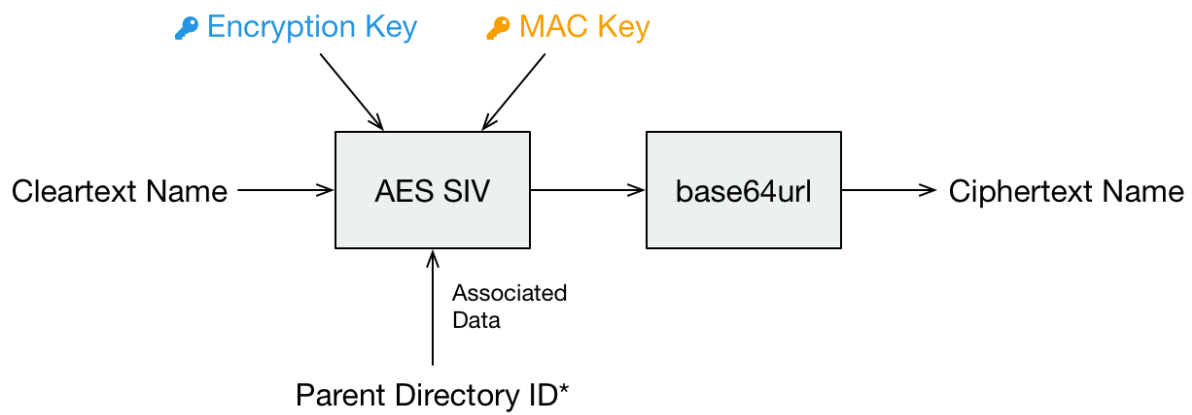
假设有以下目录结构：

```

luaCopy codeVault Root
|-- Documents
|   |-- File1.txt
|   `-- File2.txt
`-- Photos
    |-- Pic1.jpg
    `-- Pic2.jpg

```

- 假设我们要加密 "File1.txt" 文件的文件名。在加密的过程中，父文件夹的目录ID会被传递为关联数据。
1. 对于 "File1.txt" 文件，父文件夹是 "Documents"，而 "Documents" 的目录ID 是一个由系统生成的唯一标识符，比如一个随机的UUID。这个目录ID被传递给文件名加密算法作为关联数据。
 2. 在加密 "File1.txt" 文件名时，AES-SIV加密算法使用文件名、父文件夹的目录ID以及其他密钥材料来生成密文。
 3. 这样，加密后的文件名不仅仅取决于文件名本身，还取决于它所在的目录。即使相同的文件名在不同的目录下也会产生不同的加密结果，这样可以增加安全性，防止攻击者通过检测文件名的移动来破解加密。



```
ciphertextName := base64url(aesSiv(cleartextName, parentDirId, encryptionMasterKey, macMasterKey)) + '.c9r'
```

根据节点的类型，加密后的名称被用于创建文件或目录。

- 文件被存储为文件。
- 非文件被存储为目录。然后，节点的类型取决于目录内容。
 - 目录由一个名为 **dir.c9r** 的文件表示，其中包含前面叙述的目录ID。
 - 符号链接由一个名为 **symlink.c9r** 的文件表示，其中包含加密的链接目标。非文件存储为目录。节点的类型取决于目录内容。

举个栗子，一个明文目录结构如下：

```
.
├── File.txt
├── SymlinkToFile.txt
├── Subdirectory
│   └── ...
└── ...
```

经过加密之后，一个密文目录结构如下：

```
.
├── d
│   ├── BZ
│   │   └── R4VZSS5PEF7TU3PMFIM0N5GJRNBDWA
│   │       ├── 5TyvCyF255sRtfrIv**83ucADQ=.c9r # File.txt
│   │       ├── FHTa55bH*sUfVDbEb0gTL9hZ8nho.c9r # Subdirectory
│   │       │   └── dir.c9r # 包含dirId
│   │       └── gLe0GMCN358*UBf2Qk9cWCQL.c9r # SymlinkToFile.txt
│   │           └── symlink.c9r # 包含链接目标
│   └── FC
└── ...
```

```
├── ZKZRLZU0DUUYTYA4457CSBPZXB5A77 # 包含Subdirectory的内容
│   └── ...
├── masterkey.cryptomator
├── masterkey.cryptomator.DFD9B248.bkup
└── vault.cryptomator
```

Name Shortening

该层不提供额外的安全性。其唯一目的是最大限度地提高兼容性。

为了最大限度地提高兼容性，我们需要确保密文名称不超过255个字符。由于一些云同步服务可能在冲突时希望为文件添加后缀，因此cryptomator决定最多使用220个字符。

如果一个加密的名称（包括其“.c9r”扩展名）超过了这220个字符，将创建一个以其更短的SHA-1哈希和“.c9s”扩展名命名的目录。此外，还将创建一个名为“name.c9s”的反向映射文件，其中包含该目录中的原始文件。

这句话怎么理解呢？我们先来看看下面的代码

```
if (length(ciphertextName) > 220) {
    deflatedName := base64url(sha1(ciphertextName)) + '.c9s'
    inflatedNameFilePath := deflatedName + '/name.c9s'
    fileContentsPath := deflatedName + '/contents.c9r'
    symlinkFilePath := deflatedName + '/symlink.c9r'
    dirIdFilePath := deflatedName + '/dir.c9r'
}
```

这段代码的逻辑是为了确保加密后的文件名不超过220个字符，以便最大限度地提高兼容性。如果加密后的文件名（包括 .c9r 扩展名）超过了220个字符，就会采取以下步骤：

- 创建更短的标识符：** 使用 SHA-1 哈希算法对加密后的文件名进行计算，得到一个相对较短的标识符。这个标识符使用 `base64url` 编码，并附加 `.c9s` 扩展名。这个过程可以将较长的文件名映射到较短的标识符上。
- 创建目录结构：** 使用上述生成的短标识符创建一个目录，目录的名称就是短标识符（带有 `.c9s` 扩展名）。这个目录将用于存储文件的相关信息。
- 创建反向映射文件：** 在上述目录中创建一个名为 `name.c9s` 的文件，其中包含原始文件名。这样可以在需要时通过短标识符找到原始的长文件名。

节点的种类

非文件（如符号链接或目录）无论如何都将存储为目录。对于它们，没有任何变化。

但文件需要一个不同的位置来存储它们的内容。因此，我们在 `.c9s` 目录中引入了 `contents.c9r` 文件。

包含多个具有非常长名称的节点的vault可能会导致类似于下面的密文结构：

```
.
├── d
│   └── BZ
│       └── R4VZSS5PEF7TU3PMFIM0N5GJRNBDWA
│           ├── 5TyvCyF255sRtfrIv**83ucADQ=.c9r
│           └── FHTa55bH*sUfVDbEb0gTL9hZ8nho.c9r
```



```
├─ dir.c9r
├─ gLe0GMCN358*UBf2Qk9cWCQl.c9r
│   └─ symlink.c9r
├─ IjTsXtReTy6bAAuxzLPV9T0k2vg=.c9s # 缩短后的名称...
│   ├── contents.c9r # ...节点是一个常规文件
│   └─ name.c9s # ...映射到这个完整的名称
├─ q2nx5XeNCenHyQvkFD4mxYNrWpQ=.c9s # 缩短后的名称...
│   ├── dir.c9r # ...节点是一个目录
│   └─ name.c9s # ...映射到这个完整的名称
├─ u*JJCJE-T4IH-EBYASUp1u3p7mA=.c9s # 缩短后的名称...
│   ├── name.c9s # ...映射到这个完整的名称
│   └─ symlink.c9r # ...节点是一个符号链接
├─ FC
│   └─ ZKZRLZU0DUUYTYA4457CSBPZXB5A77
│       └─ ...
├─ masterkey.cryptomator
├─ masterkey.cryptomator.DFD9B248.bkup
└─ vault.cryptomator
```

Backup Directory IDs

这一层是可选的，对于 Cryptomator 加密方案的完整实现并非必需的。它不提供额外的安全性，其唯一目的是在目录文件丢失或损坏的情况下增加数据可恢复性。

通过使用包含目录ID的 `dir.c9r` 文件来混淆明文路径的层次结构，目录结构更容易受到不完整同步或位腐蚀等问题的影响。

当一个目录文件丢失或损坏时，无法计算 `dirPath`，这实际上使得虚拟文件系统中的目录内容无法访问。在理论上，可以恢复这些文件的加密内容。但由于文件名加密依赖于父文件夹的目录ID，而这个目录ID只存储在目录文件中，所有项目（文件、目录或符号链接）的名称都会丢失。

为了缓解这个问题，在创建目录时将存储一个备份目录文件。在密文目录中，将创建一个名为 `dirid.c9r` 的文件，其中包含其父文件夹的目录ID。它会像常规的密文文件一样进行加密。

「使用教程」

安装Cryptomator

[官方网站下载](#)

[Github开源网址](#)

抛开那些原理，Cryptomator 的工作机制其实很简单。就像之前所说，Cryptomator 会将在你选择的位置创建一个保险柜，而这个保险柜以文件夹的形式存在。

在你通过 Cryptomator 解锁它之后，Cryptomator 会弹出一个虚拟的磁盘，你在虚拟磁盘里的一切写入操作都会被 Cryptomator 实时加密后放入保险柜的文件夹；相反，一切读取操作（例如打开一个文件）都是由 Cryptomator 解密后展现在虚拟的磁盘里。试试看你能在保险库文件夹里找到什么东西。

必要的元素

- 保险库文件夹
- Cryptomator 客户端

创建 Vault (保险库)：

- 启动 Cryptomator 应用程序。



- 点击 "Create a new vault" (创建新的保险库)，并创建一个名称。

The screenshot shows the 'Create a new vault' dialog box. It has a light gray background. At the top, it says '为保险库创建一个名称' (Create a name for the vault). Below this is a text input field containing the word 'test'. To the right of the input field, there is a green checkmark and the text '有效的保险库名称' (Valid vault name). Below the input field, it says '保险库名称可包含以下字符：' (Vault names can contain the following characters:). There are three checkmarks followed by their respective descriptions: '字母字符 (例如 a、ж 或 𐄂)' (Alphabetic characters (e.g., a, ж, or 𐄂)), '位数' (Digits), and '连字符 (-) 或下划线 (_)' (Hyphen (-) or underscore (_)). At the bottom right, there is a green button labeled '下一步' (Next).

- 选择一个位置来保存你的保险库，设置一个密码并点击 "Create Vault" (创建保险库)。

Cryptomator 应该在哪里存储您保险库的加密文件？

☒ 自定义位置  选择...

存储位置

D:\CTF\cryptotest\test


返回

下一步

 test

 D:\CTF\cryptotest\test

☒ 启用专家设置

加密文件名的最大长度 

22

✖ 请输入 36-220 之间的数值（默认为 220）

返回

下一步

输入新密码

.....

确认新密码

.....

✓ 密码匹配!

如果丢失密码，您将无法访问您的数据。您想要创建一个恢复密钥以防您丢失密码吗？

☐ 好的，有备无患

☒ 不，谢谢。我不会丢失密码的

返回

创建保险库

✓

已添加保险库 "test"

您需要先解锁此保险库才能访问或添加内容。或者您可以在稍后任何时候再解锁它

完成

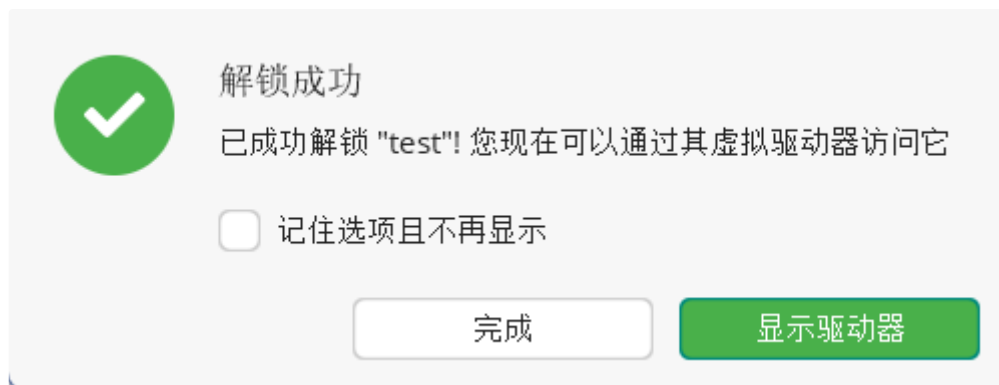
立即解锁

打开 Vault:

- 在 Cryptomator 主界面上，选择 "Unlock Vault" (解锁保险库)。



- 输入之前设置的密码并点击 "Unlock" (解锁) 。



使用 Vault:

- 解锁保险库后，你将看到一个虚拟磁盘，就像是一个常规的硬盘驱动器一样。



欢迎里面的内容

|🔒 加密卷 🔒↵

这是您的保险库的访问路径。↵

↵

任何添加到此卷的文件都将被Cryptomator 加密。您可以像在一般磁盘/文件夹上那样操作它。这只是对其内容的解密查看,您的文件会在硬盘上一直保持加密。↵

↵

您可以随时删除此文件。↵

- 将文件复制到这个虚拟磁盘中,它们将会自动被加密。

```
 NiceAsiv@DESKTOP-R64P03E → 🍄(base 3.9)
F:\ > ls

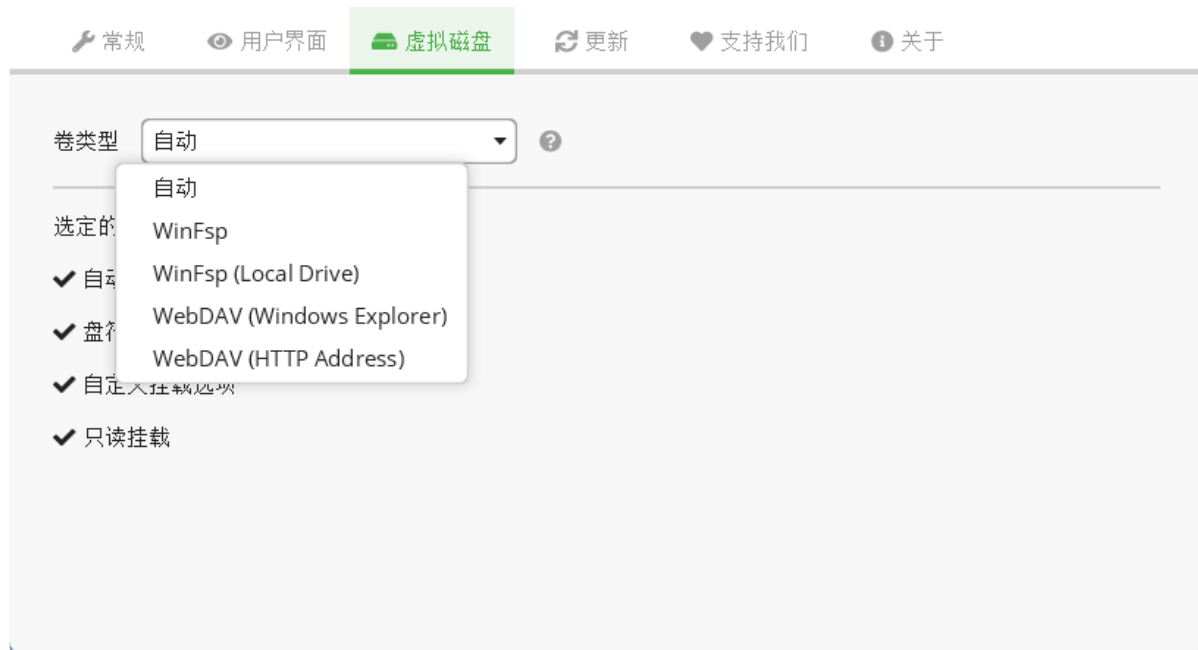
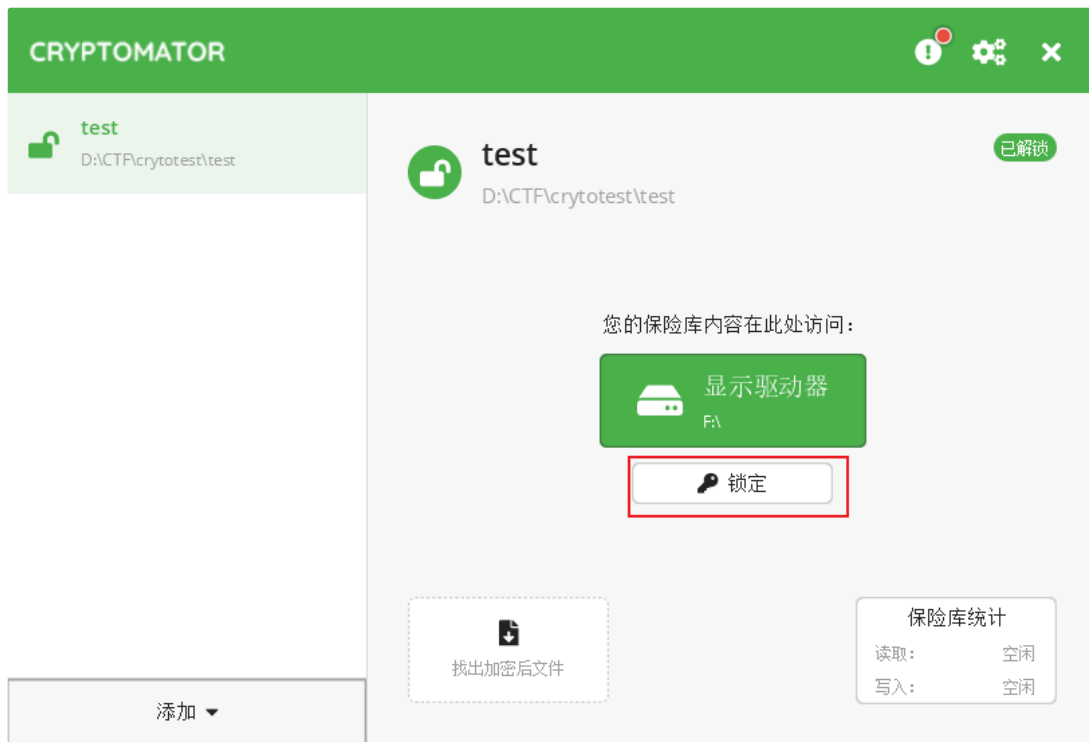
Directory: F:\

Mode                LastWriteTime         Length Name
----                -
-----                2023/11/24    23:54           820 欢迎.rtf
-----                2023/11/24    23:59            18 hello.txt
```

```
 NiceAsiv@DESKTOP-R64P03E → 🍄(base 3.9)
F:\ > tree F:\
卷 test 的文件夹 PATH 列表
卷序列号为 7D22-80D6
F:\
├─JavaCode
│   ├──.idea
│   ├──src
│   │   ├──main
│   │   │   ├──java
│   │   │   │   └─org
│   │   │   │       └─example
│   │   └──resources
│   └──test
│       └─java
└─target
    ├──classes
    │   ├──org
    │   └─example
    ├──generated-sources
    └─annotations
```

关闭 Vault:

- 在完成文件操作后,返回到 Cryptomator 主界面。
- 点击 "Lock Vault" (锁定保险库) 来关闭加密磁盘。



加密前

```
F:\> tree F:\
卷 test 的文件夹 PATH 列表
卷序列号为 7D22-80D6
F:\
├─JavaCode
│   ├──.idea
│   ├──src
│   │   ├──main
│   │   │   ├──java
│   │   │   │   └─org
│   │   │   │       └─example
│   │   └─resources
│   └─test
│       └─java
```

```

└─target
    ├──classes
    │   └─org
    │       └─example
    ├──generated-sources
    │   └─annotations

```

这时候可以发现F盘无法访问



```
D:\CTF\cryptotest > tree test
```

卷 软件盘 的文件夹 PATH 列表

卷序列号为 146F-04B1

```
D:\CTF\CRYPTOTEST\TEST
```

```

└─d
    ├──2K
    │   └─CG2CJJZENWUZXE7YNF2IICEXQQNH
    ├──2Q
    │   └─6NMDMZPG5MLAI0Q646XVDEYEUSN7KN
    │       └─RLZLzaZI_l6Avo0xmx5GwQa8oA=.c9r
    ├──3F
    │   └─Z44TP6JDQ2YHPW3EJJG63NX05N0IQD
    │       └─wNvhzQUuQEHz37_T0pivH-3faSsDrAUl.c9r
    ├──65
    │   └─ELKR26XR7YSZ3WKJZMT2XNLS5LFYCT
    │       └─K-DGnTbmsfU0UaxaDS5G9tiCkc678WWWw5xk.c9r
    ├──72
    │   └─UJ5ZTUXBKCQ23FHRKZYIQRCMAG3PRM
    │       └─e1B0ASWPF4bhU_y3ANDc5nHhPA=.c9r
    ├──EW
    │   └─3QNDMS3GR5QTA2YEEG4L05TWBFG2WP
    │       └─3gubnMamAsaYMngzF7LH11hi0zFHpUc=.c9r
    ├──GR
    │   └─BWDMEVn4IUIKMNGHRPXNDPE46J2HH
    │       ├──HTXb0dBFqX_roeSTeA7SCqR9FM=.c9r
    │       └─vASilzHvrYhXLS4TIt7mUMWgk-k=.c9r
    ├──H7
    │   └─3Y6YRU3JBTTTL6JLXWQPPUE5AMMFUX
    │       └─-mrh1RluyYEhm0EfMSqBsZemSmM=.c9r
    ├──LD
    │   └─XX2UZG4DNE23HC3WZU0I7PZLRE0Z7S
    │       ├──7kZ_QWAFvVk0ZmPj-isRHIZLCWUHcIk=.c9r
    │       └─mhFgLS_01dVaud1vITv7rxssJkHBSNKF7ufwWkFByb_.c9r
    └─NE

```



```

└─SHBSXPT2AZGSJCB7WTC07VHIE75WMM
├─QU
│   └─UDFEW5VPYEP70EKAQBETUET6P6RI3F
│       └─8bq0ajKF4_pSbthHksjgP1FafJhQvgo=.c9r
├─RC
│   └─GTCX0CI5BK3QZZFL07YG05QIWLVLN
│       └─uF9Bmai5h9r0L3vjeckdUt0BDVNBfHmww=.c9r
│           └─doLujJdPSZoo6534lyaCYD1y5Pg=.c9r
├─RL
│   └─G3WIS7FVNFRM36D3C73HSLI30FFA65
├─RQ
│   └─AZXC4RMP7YAHFPKMBZPUHLX2PIBC3W
├─SR
│   └─WX3VHPLWADH7JERJE2EV4Y2M33IKYC
├─WW
│   └─PB02U62HP3VP7YQZLTTLX3AHX2DG46
└─XW
    └─44BZMLPJV7FK53HZCS2YBHJB0F44PQ
        ├──5EqU7lpzzYYS0qRAN48QmcNAZSAz.c9r
        ├──d2-xhEl89KuV0x0BwxssLwIQoCCUIw=.c9r
        └─KLDm5ujXxv_PFYmTHnGjuWmf_Q=.c9r

```

几个关键的文件 `vault.cryptomator`

```

terkey.cryptomator  vault.cryptomator X
CTF > cryptotest > test > vault.cryptomator
eyJraWQiOiJtYXN0ZXJrZXlmaWx0m1hc3RlcmtleS5jcmlwdG9tYXRvciiIsImFsZyI6IkhTMjU2IiwidHlw

```

`masterkey.cryptomator`

```

masterkey.cryptomator X
D: > CTF > cryptotest > test > masterkey.cryptomator
1  {
2    "version": 999,
3    "scriptSalt": "5T0vg8aM1kc=",
4    "scriptCostParam": 32768,
5    "scriptBlockSize": 8,
6    "primaryMasterKey": "v1DQ67ecX+1YQ4FaNWSQtd8x7LbRHIbEzpT5sFASuUs7WHP0gaTu1Q==",
7    "hmacMasterKey": "fMKvBseYPrHK13GRovRq37U3VibSZi2LEKGl3ZKrkQz+2fJe6iJcg==",
8    "versionMac": "y+0X+whwrjqL112mDcWK7oUWTuscaYPzSkfJGEAABrQ="
9  }

```

「参考文献」

- [1] <https://juejin.cn/post/6974668741980192775>
- [2] [cryptomator/cryptomator: Multi-platform transparent client-side encryption of your files in the cloud \(github.com\)](https://github.com/cryptomator/cryptomator)
- [3] [Security Architecture – Cryptomator 1.7.0 documentation](#)