1. 错误处理不当

1.1. 问题说明

在`hmac.c`文件中，`hmac`函数对各种错误的处理方式是直接调用`printf`来输出错误信息，例如在算法无效时的处理如下：

```c
if (engine == NULL) {
    printf("invalid algorithm %s for hmac\n", algorithm);
    return NULL;
}
```

类似的，HMAC初始化、更新和最终摘要计算失败时，也都直接使用 `printf` 输出错误信息。
这种处理方式的问题在于函数本应将错误状态传递给调用者，而不是自行输出错误信息。这样设计不仅限制了调用者的错误处理方式，还使得接口的可扩展性降低，不适用于多种场景（例如在 GUI 环境中可能需要将错误信息记录到日志文件或弹出错误框，而不是直接在控制台输出）。

1.2. 改进说明

改进方法是通过返回值传递错误状态，而不是自行处理错误输出。
在修改后的代码中，通过定义不同的返回值来区分错误类型，返回`-1`表示无效算法，返回`-2`表示缓冲区不足。这样，调用者可以根据具体错误采取适合的处理措施：

```c
if (engine == NULL) {
    return -1; // 无效的算法名称
}
// test.c
else if (status == -1)
    printf("Error: invalid algorithm %s\n", algorithm);
```

在`test.c`的测试函数中，调用者可以通过检查返回值来判断错误，并根据错误类型输出适当的信息，而不是在`hmac`函数中直接决定错误的输出方式。这样设计增加了接口的灵活性和适用性。

2. 资源管理不明确

2.1. 问题说明

在 `hmac.c` 文件的 `hmac` 函数中，函数内动态分配了输出摘要字符串所需的内存，调用者需要在使用后自行释放这块内存。但是，接口定义中并没有显式释放内存，容易导致调用者在使用该函数时忽视内存管理，从而造成内存泄漏。
并且，在 `test.c` 文件中，我发现了疑似想要释放该内存的代码，这也是不合适的。
以下是原代码中的问题部分：

```c
// hamc.c
char* result = (char*)malloc(output_length * 2 + 1);

// test.c
char* result = hmac(src, strlen(src), key, strlen(key), algorithm);
if(result != NULL)
    delete result;
```

这种隐含的内存分配方式，增加了调用者管理资源的难度，特别是在复杂的应用场景下，容易造成资源管理混乱。

2.2. 改进说明

为避免内存管理不清晰的问题，我选择让调用者提供存放结果的缓冲区，从而明确资源的分配和释放责任：调用者需传入足够大小的缓冲区 `out_buffer`，且函数在该缓冲区内存放计算结果，确保了资源管理的清晰和高效性：

```c
// hmac.h
int hmac(const char* str, int str_length, const char* key,
         int key_length, const char* algorithm,
         char* out_buffer, int buffer_size);

// test.c
char result[EVP_MAX_MD_SIZE * 2 + 1]; // 确保缓冲区足够大
```

这样，`hmac`函数仅负责将结果写入传入的缓冲区，无需在函数内动态分配内存，调用者也不再需要手动释放内存。

3. 内存分配的灵活性不足

3 .1. 问题说明

在原始实现中，`hmac` 函数内部总是动态分配内存用于存储返回的摘要字符串，并将该内存返回给调用者。这样不仅造成资源管理的不明确，而且每次调用时都增加了额外的内存分配开销，尤其是在多次调用的情况下，性能损耗更为明显。

```c
char* result = (char*)malloc(output_length * 2 + 1);
```

在一些场景下，调用者可能希望复用已分配的缓冲区，从而减少内存分配和释放的操作，提高性能，但原始接口没有提供这种灵活性。

3.2. 改进说明

此问题的改进与上一问题的改进属于共同的改进。让调用者提供存储结果的缓冲区 `out_buffer`，并指定其大小 `buffer_size`。在 `hmac` 函数内部，先检查缓冲区大小是否足够（至少是 `EVP_MAX_MD_SIZE * 2 + 1`），如果不足则返回错误码 `-2`，以便调用者知晓并根据需求调整缓冲区大小。
这样设计的接口使调用者可以复用缓冲区，减少内存分配和释放的频率，提升性能：

```c
if (buffer_size < output_length * 2 + 1) {
    return -2; // 缓冲区不足
}
```

这种改进让接口在保持安全的前提下变得更加灵活和高效，尤其适用于高性能或嵌入式场景。