COMP2700 实验室 10 - 哈希和 MAC

本教程/实验中的部分练习选自 Paar & Pelzl 的《理解密码学》（第 11 章和第 12 章）。

练习 3 和练习 5 需要两个 python 脚本，可以从课程的 Wattle 页面下载 (lab10-code.zip)，也可以使用以下命令直接下载到终端：

```
$ wget http://users.cecs.anu.edu.au/~tiu/comp2700/lab10-code.zip
```

练习 1.加密散列函数的早期应用之一是在计算机系统中存储用于用户身份验证的密码。这种方法是在输入密码后对密码进行散列，然后与存储的（散列）参考密码进行比较。人们很早就意识到，只存储密码的散列版本就足够了。

假设你是一名黑客，可以访问散列密码列表。当然，你想从列表中恢复密码来冒充某些用户。请讨论以下三种攻击中哪一种可以实现这一点。准确描述每种攻击的后果：

- 攻击 A：你可以破解 h 的单向属性。
- 攻击 B：你可以找到 h 的第二个预图像。
- 攻击 C： 你可以找到 h 的碰撞。

练习 2.我们考虑三种不同的散列函数，它们分别产生长度为 64、128 和 160 位的输出。经过大约多少次随机输入后，发生碰撞的概率为 ε =0.5？

练习 3.考虑下面这个在 ECB 模式下使用 AES 构建的散列函数：给定长度为 n 块的输入x，其中每个块的长度为 128 位（16 字节），散列函数的输出是通过首先加密x，然后计算密文所有 128 位块的 XOR 结果得到的。更准确地说，如果 $x = x_1 \cdots x_n$，其中每个 x_i 恰好是一个块（128 位长），那么哈希值为

$$H(x) = e_k\,(x_1\,) \oplus e_k\,(x_2\,) \oplus \cdots \oplus e_k\,(x\,)_n$$

其中，$\oplus$ 是 XOR 函数，k 是一个固定的 128 位密钥。请注意，这里的加密密钥并不是为了保护输入信息的机密性；它是哈希算法的一部分，因此其值需要公开。

所提供的 python 脚本 ecbhash.py 提供了这一散列函数的简单实现。使用 echbhash.py 脚

本计算以下文本的哈希值：

```
0123456789ABCDEF0123456789abcdef
```

然后构建该文本的第二个预图像。下面提供了一些有用的 python 命令来帮助您解决这个问题。

在本练习中，我们将使用 python 以交互方式执行一些（字节）字符串操作。

要使用 ecbhash.py 脚本，请使用 "import "命令在 python shell 中加载它。哈希函数名为 ecbhash；调用时使用 "ecbhash.ecbhash(input)"，其中 "input "为字节字符串。

要构建一个字节字符串样本，最简单的方法是在字符串前使用前缀 "b"--这会导致 python 将字符串解释为字节序列。下面是一个 Python 会话示例，用于计算输入的哈希值（假设在 Linux 终端运行）：

```
alice@comp2700-lab:~/lab10$ python3
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
输入 "help"、"copyright"、"credits "或 "license "获取更多信息。
>>> 导入 ecbhash
>>>
>>> test=b'这是一个测试！！'
>>> ecbhash.ecbhash(test)
"9a23fcc53603ee56c2179f2156b2cc5e
>>>
```

要访问字节字符串中的字节，可以使用范围运算符，例如，x[i:j] 表示字节 x[i]、x[i+1]、...、x[j-1] 的序列。操作符 "+"被重载，允许连接两个字节字符串。下面是这些运算符的一些示例：

```
>>> x=b'01234567'
>>> x[0:4]
b'0123'
>>> x[4:8]
b'4567'
>>> x[4:8]+x[0:4]
b'45670123'
>>>
```

练习 4.某些基于块密码的加密操作（如 CBC 加密、散列或 MAC 计算）要求输入信息的大小是块大小的倍数（例如，对于 AES，要求是 16 字节的倍数）。如果输入信息的大小不是数据块大小的倍数，则需要在处理输入信息前添加一些填充。PKCS#7 填充[1] 方案是一种广泛使用的填充方案，用于填充信息，使其长度成为区块大小的倍数。

这种填充方案的工作原理如下：假设数据块大小为 b 字节。给定输入值 m，如果 m 的最后一个数据块的长度为 $(b-r)$ 字节，则将 r 的字节值添加到 m 中，直到最后一个数据块的长度为 $(b-r)$ 字节。

---

[1] 有关 PKCS#7 的规范，请访问 https://www.ietf.org/rfc/rfc2315.txt 。

的长度为 b 字节。例如，假设 m 是以下字节序列（此处使用 python 符号）：

b'12345'

和数据块大小 $b = 16$。因此，m 是一个 5 字节长的字节数组，比块大小（16 字节）少 11 个字节。在这种情况下，r=11（或用 HEX 表示的 0xb），因此我们在字节数组 m 中填充 11 个字节，每个字节都包含值 r (0xb)。因此，填充后的 m（用 python 字节符号表示）为

b'12345\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b'

pycryptodome 库（Crypto.Util.Padding）提供了一个名为 **pad** 的函数，用于实现 PKCS#7 填充方案。pad 函数有两个参数：输入字节和块大小（以字节为单位）。还有一个相应的 **unpad** 函数，可以将填充后的信息反转为原始信息。下面是一个 python 会话示例，用于计算上例中块大小为 16 字节的 m 的填充版本：

```
>>> 从 Crypto.Util.Padding 导入 *
>>> m=b'12345'
>>> s=pad(m,16)
>>> s
b'12345\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b'
>>> unpad(s,16)
b'12345'
>>>
```

在 PKCS#7 标准的官方规范中，如果信息长度m已经是块大小的倍数，则会增加一个额外的块（该块的每个字节都包含块大小的字节值）。您可以使用 pycryptodome 的 pad 函数确认这一点。假设我们修改了 PKCS#7 填充，如果输入值已经是数据块大小的倍数，则不添加填充。您认为这种修改后的填充方案会有什么问题？

练习 5.在 SHA-1 中，即使输入信息的长度是 SHA-1 数据块大小的倍数，输入到散列函数的信息也总是被填充的。为什么会出现这种情况？请根据所应对的攻击说明答案。

练习 6.在本练习中，我们将实现 MAC 讲座中讨论的对 CBC-MAC 的伪造攻击。我们将使用基于 128 位密钥 AES 的 CBC-MAC 实现（参见提供的 cbcmac.py 脚本）。在本练习中，我们将在交互模式下使用 python，因为它更易于操作字节串和转换字符编码。

我们首先运行 python3 并导入 cbcmac.py 脚本。我们在此展示在实验室虚拟机中运行的交互式 python 会话的输出结果：

```
alice@comp2700-lab:~/lab10$ python3
Python 3.6.9 (default, Jul 17 2020, 12:50:27)
[GCC 8.4.0] on linux
输入 "help"、"copyright"、"credits "或 "license "获取更多信息。
>>> 导入 cbcmac
```

然后，我们定义要计算 MAC 的输入文本和 MAC 密钥：

```
>>> input=b'测试 CBC-MAC! '
>>> key=b'0123456789abcdef'
>>>
```

注意：前缀 b 表示 python 应将后面的字符串视为字节序列，而不是字符串。在 python 中，其类型为 "字节"。

在本练习中，我们将 IV 固定为以下值：b'fedcba9876543210'（这是 cbcmac.py 文件中的硬编码）。

(密钥和 IV 的值不必完全相同，只要各为 16 字节，任何值都可以）。

接下来，我们使用 cbcmac.py 中的 gen_mac 函数计算 "输入 "与密钥 "key "的 MAC 值：

```
>>> mac=cbcmac.gen_mac(input,key)
>>> mac
'0a607906c6a1173a4655344f11a48b5b'
>>>
```

请注意，MAC 输出是以 HEX 字符串形式给出的。

在密钥 "key "的情况下，我们可以使用 verify_mac 函数来验证 "mac "是否是 "input "的正确 MAC：

```
>>> cbcmac.verify_mac(input, key, mac)
True
>>>
```

你现在的任务是构建一个不同的输入，称之为 "fake_input"，这样 fake_input 的 MAC 就正好是 "mac"，而无需使用上面给出的密钥。

提示：查看 CBC-MAC 讲座，了解攻击的原理。这里有几个 Python 命令，你可能会觉得有用：

-要连接两个字节字符串，可以使用运算符 "+"。例如

```
>>> x=b'0123'
>>> y=b'4567'
>>> x+y
b'01234567'
```

-要将 HEX 字符串转换为字节，请使用 "bytes.fromhex(<string>)"。例如

```
>>> str='414243'
>>> bytes.fromhex(str)
b'ABC'
>>>
```

-要计算两个字节字符串的 xor 值，请使用 strxor。为此，您需要导入 Crypto.Util.strxor（参见 Lab 9）。

扩展练习（可选）

练习 7.对于散列函数来说，拥有足够大的输出比特数（例如 160 比特）对于挫败基于生日悖论的攻击至关重要。为什么更短的输出长度（例如 80 比特）对 MAC 来说就足够了呢？为了回答这个问题，请假设通过信道以明文发送信息x及其 MAC：$(x, MAC_k(x))$。确切地说明攻击者要如何攻击这个系统。

练习 8.我们研究两种用加密保护完整性的方法。

a) 假设我们采用了一种加密和完整性保护相结合的技术，其中密文 c 的计算公式为

$$c = e\_k(x \parallel \text{anticipated}(x))$$

其中 导() 是哈希函数。如果攻击者知道与密文c相对应的全部明文x，那么这种技术就不适用于流密码加密。请解释攻击者如何用任意x'替换x，并计算出c'，从而使接收者能正确验证信息。假设 x 和 x' 的长度相等。

b) 如果使用 MAC 计算校验和，攻击是否仍然适用？

$$c = e_{k1}\ (x \parallel MAC_{k2}\ (x))$$

假设$e(\ )$是上述流密码。

练习 9.在本练习中，我们将测试 Google 和 CWI 发现的 SHA-1 碰撞攻击。为了演示碰撞攻击，我们将创建两个内容不同但 SHA-1 哈希值相同的 PDF 文档。要计算文件的 SHA-1 哈希值，可以使用 linux 中的 "shasum "命令。

碰撞攻击仅适用于一种非常特殊的 PDF 文档，即包含嵌入式 JPEG 图像的 PDF 文档。为了创建这些 PDF，我们将使用 Word 文档，首先创建两个大小几乎相同但内容略有不同的 PDF 文档。您可以使用提供的 Word 模板（letter.docx）--改变该文档中的美元值来创建两个不同的文档，并将它们导出为 PDF 格式。此攻击仅适用于大小小于 64kb 的 PDF 文档，因此如果您生成的 PDF 文件大于 64kb，则需要缩小其大小。

我们将使用此处提供的漏洞利用脚本 https://github.com/nneonneo/sha1collider 下载漏洞

利用脚本并运行

./collide.py file1.pdf file2.pdf -progressive

其中 file1.pdf 和 file2.pdf 是要生成 SHA-1 碰撞的 PDF 文件。如果成功，就会生成两个文件 out-file1.pdf 和 out-file2.pdf。使用 shasum 确认它们具有相同的 SHA-1 哈希值。

请注意，如果脚本抱怨缺少 "cjpeg"，您可能需要安装 libjpeg-progs 软件包：

sudo apt install libjpeg-progs