- 服务器允许在一小段时间内使用同一个随机数；
296 - 客户端和服务器使用同步的、可预测的随机数生成算法。

### 1. 预先生成下一个随机数

可以在 Authentication-Info 成功首部中将下一个随机数预先提供给客户端。这个首部是与前一次成功认证的 200 OK 响应一同发送的。

```
Authentication-Info: nextnonce="<nonce-value>"
```

有了下一个随机数，客户端就可以预先发布 Authorization 首部了。

尽管这种预授权机制避免了请求 / 质询循环（加快了事务处理的速度），但实际上它也破坏了对同一台服务器的多条请求进行管道化的功能，因为在发布下一条请求之前，一定要收到下一个随机值才行。而管道化是避免延迟的一项基本技术，所以这样可能会造成很大的性能损失。

### 2. 受限的随机数重用机制

另一种方法不是预先生成随机数序列，而是在有限的次数内重用随机数。比如，服务器可能允许将某个随机数重用 5 次，或者重用 10 秒。

在这种情况下，客户端可以随意发布带有 Authorization 首部的请求，而且由于随机数是事先知道的，所以还可以对请求进行管道化。随机数过期时，服务器要向客户端发送 401 Unauthorized 质询，并设置 WWW-Authenticate:stale=true 指令：

```
WWW-Authenticate: Digest
    realm="<realm-value>"
    nonce="<nonce-value>"
    stale=true
```

重用随机数使得攻击者更容易成功地实行重放攻击。虽然这确实降低了安全性，但重用的随机数的生存期是可控的（从严格禁止重用到较长时间的重用），所以应该可以在安全和性能间找到平衡。

此外，还可以通过其他一些特性使重放攻击变得更加困难，其中就包括增量计数器和 IP 地址测试。但这些技术只能使攻击的实施更加麻烦，并不能消除由此带来的安全隐患。

### 3. 同步生成随机数

还可以采用时间同步的随机数生成算法，客户端和服务器可根据共享的密钥，生成第三方无法轻易预测的、相同的随机数序列（比如安全 ID 卡）。

297 这些算法都超出了摘要认证规范的范畴。