

Padding Oracle Attack 详解

关于 Padding Oracle Attack , 最好的文章也许要算

[http://www.gdssecurity.com/l/b/2010/09/14/automated-padding-oracle-attacks-wit](http://www.gdssecurity.com/l/b/2010/09/14/automated-padding-oracle-attacks-with-padbuster/)

[h-padbuster/](http://www.gdssecurity.com/l/b/2010/09/14/automated-padding-oracle-attacks-with-padbuster/)了。我只是以这篇文章为基础写一版中文的介绍文章，并加上我自

己的一些想法重新组织资料调整结构，希望能够降低门槛让更多的人看懂。由于

涉及到一些加密的概念，而加密我并没有深入了解过，所以这里只写出需要用到

那一小部分，估计还是难免会有一些错误，☺。

只知道攻击而不知道原理无疑是可悲的。

云舒 (wustyunshu@hotmail.com)

2010 年 9 月 28 日 , Ph4nt0m、80sec

一、 背景知识

1.1 分组密码和填充

常用的对称加密算法 ,如 3DES、AES 在加密时一般使用分组密码(Block Cipher)。将明文以数据块 (Block) 为单位进行加密 , 常见的如 64 bit、128 bit、256 bit , 而不是每次只加密一个比特。

这样就带来一个问题 , 数据的长度不可能恰好是 block 的整数倍。较长的数据就涉及到分组操作 , 不能整除的剩余部分数据就涉及到填充操作。分组比较简单直接按照 bit 切分即可 , 填充方法最常见的是 PKCS#5 , 在最后一个 block 中将不足的 bit 位数以 bit 为值填充。最后一个 block 缺少 5 bit , 则填充 5 个 0x05 到 block 结尾 , 缺少 2 bit 则填充 2 个 0x02 到结尾。以 64 bit 的 block 举例 :

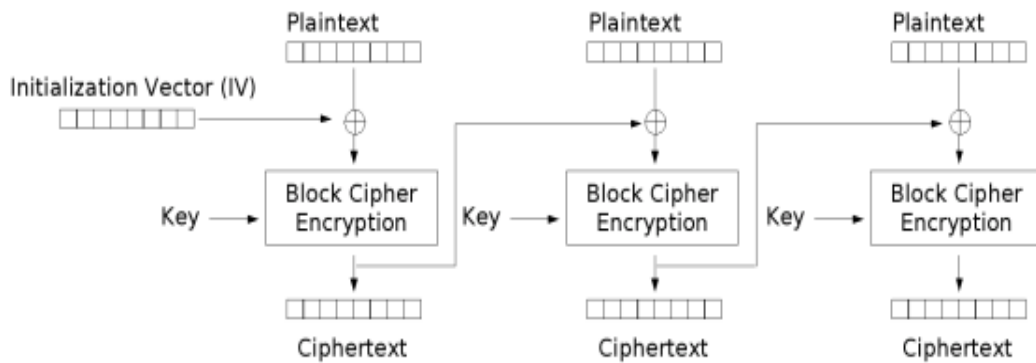
	BLOCK #1								BLOCK #2							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Ex 1	F	I	G													
Ex 1 (Padded)	F	I	G	0x05	0x05	0x05	0x05	0x05								
Ex 2	B	A	N	A	N	A										
Ex 2 (Padded)	B	A	N	A	N	A	0x02	0x02								
Ex 3	A	V	O	C	A	D	O									
Ex 3 (Padded)	A	V	O	C	A	D	O	0x01								
Ex 4	P	L	A	N	T	A	I	N								
Ex 4 (Padded)	P	L	A	N	T	A	I	N	0x08	0x08	0x08	0x08	0x08	0x08	0x08	0x08
Ex 5	P	A	S	S	I	O	N	F	R	U	I	T				
Ex 5 (Padded)	P	A	S	S	I	O	N	F	R	U	I	T	0x04	0x04	0x04	0x04

1.2 分组密码加密的模式

上一节描述了如何将明文按照指定大小的 block 进行分组，如何将位数不足的 block 填充补足，现在就可以开始加密了。分组密码加密有四种模式，分别是 ECB、CBC、CFB 和 OFB，其中 CBC 是 IPSEC 的标准做法，这里主要介绍。

CBC(Cipher Block Chaining)引入了一个随机的初始化向量(initialization vector)来加强密文的随机性，保证相同的明文多次加密都得到不一样的密文。明文先与初始化向量做 XOR 操作，然后按照加密算法加密。同时，上一个 block 加密后的密文作为下一个 block 的初始化向量，用来保证更强的安全性。需要注意的是，这个初始化向量会带在密文当中，否则解密端不知道初始化向量也无法解密密文。

加密过程如下图：



Cipher Block Chaining (CBC) mode encryption

二、 Padding Oracle Attack

2.1 问题的原因

明文分组和填充就是 Padding Oracle Attack 的根源所在,但是这些需要一个前提,那就是应用程序对异常的处理。当提交的加密后的数据中出现错误的填充信息时,不够健壮的应用程序解密时报错,直接抛出“填充错误”异常信息。

攻击者就是利用这个异常来做一些事情,假设有这样一个场景,一个 WEB 程序接受一个加密后的字符串作为参数,这个参数包含用户名、公司 ID 和角色 ID。参数加密使用的最安全的 CBC 模式,每一个 block 有一个初始化向量。当提交参数时,服务端的返回结果会有下面 3 种情况:

- a. 参数是一串正确的密文,分组、填充、加密都是对的,包含的内容也是正确的,那么服务端解密、检测用户权限都没有问题,返回 HTTP 200。
- b. 参数是一串错误的密文,包含不正确的 bit 填充,那么服务端解密时就会抛出异常,返回 HTTP 500 server error。
- c. 参数是一串正确的密文,包含的用户名是错误的,那么服务端解密之后检测权限不通过,但是依旧会返回 HTTP 200 或者 HTTP 302,而不是 HTTP 500。

攻击者无需关心用户名是否正确,只需要提交错误的密文,根据 HTTP Code 即可做出攻击。根据应用程序的状态码在不知道密钥的情况下一个 bit 一个 bit 的猜解出密文对应的明文,也可以伪造出任意明文加密后的密文。

2.2 根据密文猜解明文

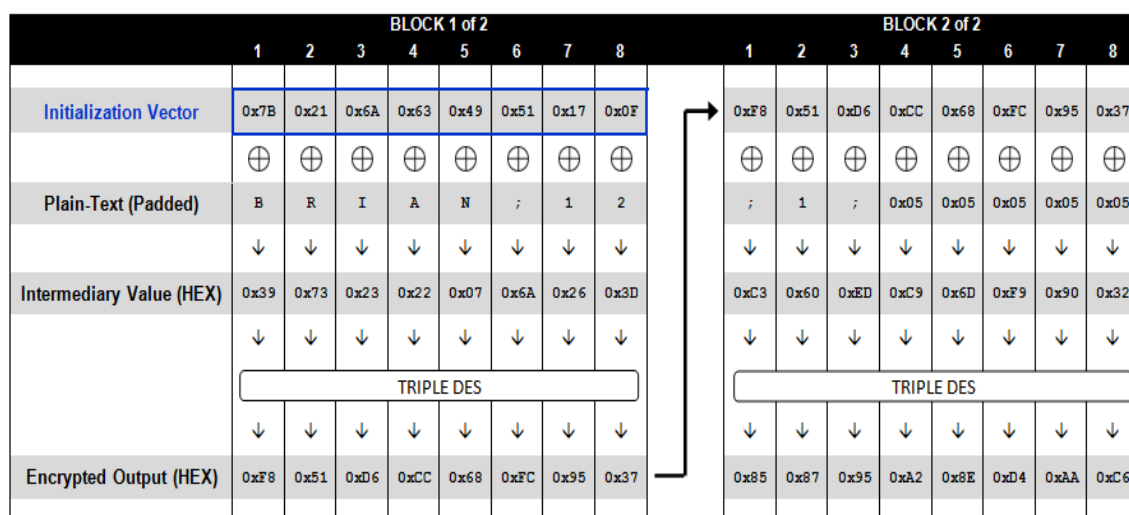
假设有这样一个应用,

<http://sampleapp/home.jsp?UID=7B216A634951170FF851D6CC68FC9537858795A2>

8ED4AAC6，我们来看看在不知道明文的情况下，如何破解出明文。首先将密文分组，前面 8 字节是初始化向量，后面的 16 字节就是加密后的数据，如下图(为了更清晰的说明加密解密的变化过程，这里把明文标出了)：

	INITIALIZATION VECTOR								BLOCK 1 of 2								BLOCK 2 of 2							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Plain-Text	-	-	-	-	-	-	-	-	B	R	I	A	N	;	1	2	;	1	;					
Plain-Text (Padded)	-	-	-	-	-	-	-	-	B	R	I	A	N	;	1	2	;	1	;	0x05	0x05	0x05	0x05	0x05
Encrypted Value (HEX)	0x7B	0x21	0x6A	0x63	0x49	0x51	0x17	0x0F	0xF8	0x51	0xD6	0xCC	0x68	0xFC	0x95	0x37	0x85	0x87	0x95	0xA2	0x8E	0xD4	0xAA	0xC6

根据上面分组密码加密模式中的加密图示，我们先看看 BARIN;12;1 是如何被加密的。



类似的，解密只不过是一个逆向的过程，如下图：

	BLOCK 1 of 2								BLOCK 2 of 2							
	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Encrypted Input (HEX)	0xF8	0x51	0xD6	0xCC	0x68	0xFC	0x95	0x37	0x85	0x87	0x95	0xA2	0x8E	0xD4	0xAA	0xC6
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
	TRIPLE DES								TRIPLE DES							
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Intermediary Value (HEX)	0x39	0x73	0x23	0x22	0x07	0x6a	0x26	0x3D	0xC3	0x60	0xED	0xC9	0x6D	0xF9	0x90	0x32
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Initialization Vector	0x7B	0x21	0x6A	0x63	0x49	0x51	0x17	0x0F	0xF8	0x51	0xD6	0xCC	0x68	0xFC	0x95	0x37
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Plain-Text (Padded)	B	R	I	A	N	;	1	2	;	1	;	0x05	0x05	0x05	0x05	0x05

VALID PADDING

现在开始破解了，我们先将初始化向量设置为 0，提交如下请求，

<http://sampleapp/home.jsp?UID=0000000000000000F851D6CC68FC9537>，服务器势必返回一个 HTTP 500 的服务器错误信息，因为对第一个 block 解密的时候 XOR 初始化向量得到最后一个字节是 0x3D，这显然是一个错误的填充。这时候解密示意图如下：

	BLOCK 1 of 1							
	1	2	3	4	5	6	7	8
Encrypted Input	0xF8	0x51	0xD6	0xCC	0x68	0xFC	0x95	0x37
	↓	↓	↓	↓	↓	↓	↓	↓
	TRIPLE DES							
	↓	↓	↓	↓	↓	↓	↓	↓
Intermediary Value	0x39	0x73	0x23	0x22	0x07	0x6a	0x26	0x3D
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Initialization Vector	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00
	↓	↓	↓	↓	↓	↓	↓	↓
Decrypted Value	0x39	0x73	0x23	0x22	0x07	0x6a	0x26	0x3D

X

INVALID PADDING

现在我们在初始化向量全 0 的情况下变化最后一个字节为 0x01，提交请求

http://sampleapp/home.jsp?UID=000000000000000001F851D6CC68FC9537，如上次

请求一样服务器继续返回一个 HTTP 500 的服务器错误信息，解密示意图如下：

BLOCK 1 of 1								
	1	2	3	4	5	6	7	8
Encrypted Input	0xF8	0x51	0xD6	0xCC	0x68	0xFC	0x95	0x37
	↓	↓	↓	↓	↓	↓	↓	↓
TRIPLE DES								
	↓	↓	↓	↓	↓	↓	↓	↓
Intermediary Value	0x39	0x73	0x23	0x22	0x07	0x6a	0x26	0x3D
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Initialization Vector	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x01
	↓	↓	↓	↓	↓	↓	↓	↓
Decrypted Value	0x39	0x73	0x23	0x22	0x07	0x6a	0x26	0x3C

X

INVALID PADDING

我们将最后一个字节从 0x00 逐渐往 0xFF 方向递增，迟早会猜测到一个正确的值，

让填充的最后一个字节是 0x01，成为一个正确的 padding。当初始化向量是

00000000000000003C 的时候，成功了，服务器返回 HTTP 200，解密示意图如下：

Block 1 of 1								
	1	2	3	4	5	6	7	8
Encrypted Input	0xF8	0x51	0xD6	0xCC	0x68	0xFC	0x95	0x37
	↓	↓	↓	↓	↓	↓	↓	↓
TRIPLE DES								
	↓	↓	↓	↓	↓	↓	↓	↓
Intermediary Value	0x39	0x73	0x23	0x22	0x07	0x6a	0x26	0x3D
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Initialization Vector	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x3C
	↓	↓	↓	↓	↓	↓	↓	↓
Decrypted Value	0x39	0x73	0x23	0x22	0x07	0x6a	0x26	0x01

✓

VALID PADDING

根据 XOR 的交换性可以知道解密时一个临时中间值(Intermediary Value)最后一个字节是 0x3D，因此可以预测当初始化向量最后一个字节是 0x3F 的时候，padding 的最后一个字节会是 0x02。现在将初始化向量的最后一个字节设置为 0x3F，变化其倒数第二个字节，开始猜解临时中间值的倒数第二个字节。猜解成功时示意图如下：

	1	2	3	4	5	6	7	8
Encrypted Input	0xF8	0x51	0xD6	0xCC	0x68	0xFC	0x95	0x37
	↓	↓	↓	↓	↓	↓	↓	↓
	TRIPLE DES							
	↓	↓	↓	↓	↓	↓	↓	↓
Intermediary Value	0x39	0x73	0x23	0x22	0x07	0x6a	0x26	0x3D
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Initialization Vector	0x00	0x00	0x00	0x00	0x00	0x00	0x24	0x3F
	↓	↓	↓	↓	↓	↓	↓	↓
Decrypted Value	0x39	0x73	0x23	0x22	0x07	0x26	0x02	0x02

VALID PADDING ✓

以此类推，我们可以猜解出完整的临时中间值。

	1	2	3	4	5	6	7	8
Encrypted Input	0xF8	0x51	0xD6	0xCC	0x68	0xFC	0x95	0x37
	↓	↓	↓	↓	↓	↓	↓	↓
	TRIPLE DES							
	↓	↓	↓	↓	↓	↓	↓	↓
Intermediary Value	0x39	0x73	0x23	0x22	0x07	0x6a	0x26	0x3D
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Initialization Vector	0x31	0x7B	0x2B	0x2A	0x0F	0x62	0x2E	0x35
	↓	↓	↓	↓	↓	↓	↓	↓
Decrypted Value	0x08	0x08	0x08	0x08	0x08	0x08	0x08	0x08

VALID PADDING ✓


同样基于 XOR 的交换性,我们将临时中间值(0x39 0x73 0x23 0x22 0x07 0x6A 0x26 0x3D)与我们要猜解的被加密的串 (7B216A634951170F) 做 XOR 即可得到明文的第一个 block 即 BRIAN;12

同样的方式,即可猜解密文的第二个 block 的明文。需要注意的是,第一个 block 的密文 F851D6CC68FC9537 将是第二个密文的初始化向量。推而广之,前一个 block 的密文是后一个 block 的初始化向量。

2.3 伪造指定内容的密文

伪造指定内容的密文,才是真正攻击的开始。我们可以伪造 cookie, 伪造 WAP 站点的 SID 等等内容。在上一节中,我们知道密文 F851D6CC68FC9537 经过解密 (3DES 还是 AES? 密钥是多少? 你还问这个,我们需要这个么?)后的临时中间值是 39732322076A263D, 那么我们只需要传递指定的初始化向量,就能 XOR 出我们想要的任何值。比如说加密 TEST, 示意图如下:

	1	2	3	4	5	6	7	8
Encrypted Input	0xF8	0x51	0xD6	0xCC	0x68	0xFC	0x95	0x37
	↓	↓	↓	↓	↓	↓	↓	↓
	TRIPLE DES							
	↓	↓	↓	↓	↓	↓	↓	↓
Intermediary Value	0x39	0x73	0x23	0x22	0x07	0x6A	0x26	0x3D
	⊕	⊕	⊕	⊕	⊕	⊕	⊕	⊕
Initialization Vector	0x6D	0x36	0x70	0x76	0x03	0x6E	0x22	0x39
	↓	↓	↓	↓	↓	↓	↓	↓
Decrypted Value	T	E	S	T	0x04	0x04	0x04	0x04



也就是说，我们只需要将 6D367076036E2239 作为密文的初始化向量，F851D6CC68FC9537 作为密文提交上去，服务端解密后必然解密出 TEST，后面加 4 字节的填充。

如果更长的密文该怎么伪造了？因为初始化向量每个 block 都不同，而且前一个 block 会影响后一个 block，因此我们需要先将明文分组，从最后一个 block 算起。至于具体的计算，就不说了，稍微想想就能明白。

2.4 .NET 框架的问题

这个漏洞在 <http://sd.csdn.net/a/20100926/279861.html> 有详细的描述，只是作者精通 .NET 开发而不懂安全，所以没有办法描述为什么会这样。说到底，下载 web.config 不过就是一个伪造制定内容的密文的问题。

这是一种攻击思路，而不应该局限为一种手段，应该有很多应用会被类似的攻击所威胁。