정보보호 Project 2 Report

2017150422 곽진성

1. Hash 함수 취약점 분석

H(m, t)

1. A_0= (m[0], m[1], …, m[15])
2. B_0= (m[16], m[17], …, m[31])
3. For $i = 0, 1, \ldots, t-2$:
4. $rb_i = (B_i[0], B_i[1], B_i[2], B_i[3])$
5. $A_{i+1} = Round(rb_i, A_i)$
6. $B_{i+1} = Round((\mathrm{i}, \mathrm{i}+1, \mathrm{i}+2, \mathrm{i}+3), A_i)$
7. End For
8. $rb_{t-1} = (B_{t-1}[0], B_{t-1}[1], B_{t-1}[2], B_{t-1}[3])$
9. A_t= $Round'(rb_{t-1}, A_{t-1})$
10. $O = A_t \oplus A_0 \oplus B_0$
    (which is $A_t[0] \oplus A_0[0] \oplus B_0[0], \ldots, A_t[15] \oplus A_0[15] \oplus B_0[15]$)

$\boldsymbol{Y = Round(rb, X)}$

Each of rb and X is a 32-bit value which can be represented by rb = (rb[0], rb[1], rb[2], rb[3]) and X = (X[0], X[1], … , X[15]).

Each of rb[i] and X[i] is a unit vector of which size is 8-bit.

1. $(Y[0], \ldots, Y[3]) = F(rb, (X[0], \ldots, X[3])) \oplus (X[4], \ldots, X[7])$
2. $(Y[4], \ldots, Y[7]) = (X[8], \ldots, X[11])$
3. $(Y[8], \ldots, Y[11]) = (X[12], \ldots, X[15])$
4. $(Y[12], \ldots, Y[15]) = (X[0], \ldots, X[3])$

1. $p[i] = X[i] \oplus rk[i] \quad (for\ i = 0,1,2,3)$
2. $q[i] = S(p[i]) \quad (for\ i = 0,1,2,3)$ // AES S-box function
3. $y^T = M \cdot q^T$ // AES MixColumn function

위 Hash 함수에는 다음과 같은 취약점이 있습니다. 취약점들을 이용하면, 특정 패턴을 갖는 message에 대해 쉽게 collision pair를 만들 수 있습니다.

문제에서 주어진 Hash 함수의 취약점은, B_0에 대해, 첫 4bytes를 제외하고는 의존적이지 않다는 점입니다. B_1부터는 A를 기반으로 만들기 때문에, B_0가 A_t를 만들기 위해 사용되는 부분은 첫 4bytes가 다입니다. 따라서 rb값으로 쓰이지 않는 뒤의 12bytes는 임의로 마지막 xor연산이 원하는 hash값을 만들어내도록 조정하기 위해 쓸 수 있습니다.

Round Number t가 5인 점은, 위 취약점을 이용할 수 있게 합니다. 처음 Round를 통해 계산된 A1의 첫 4byte는 A5가 되면 다시 원래 자리로 돌아옵니다. 즉,

$A_5[0:4] = A_1[0:4] = F(B_0[0:4], A_0[0:4]) \oplus A_0[4:8]$ (python식 indexing, 끝 수는 미포함) 으로 나타낼 수 있습니다.

F함수가 갖는 취약점은 같은 P값을 만드는 X와 rb pair에 대해 같은 결과값 y를 산출해낸다는 점입니다. P는 X와 rb의 xor연산을 통해 만들어진다는 점을 이용해, A_0, B_0의 첫 4byte를 사용하는 첫 번째 round function에서 F의 결과를 A, B를 적당히 조절해 원하는 대로 만들어낼 수 있습니다. 여기에 xor되는 $A_0[4:8]$을 0x00으로 설정하면, $A_5[0:4] = F_{result}$ 가 되게 할 수 있습니다.

Collision Pair를 더 간단하게 찾을 수 있도록 도움이 되는 성질은 $O = A_t \oplus A_0 \oplus B_0$ 입니다. $A_0 = B_0$이도록 M을 설정하면, A_t가 곧 O가 됩니다.

위 취약점들을 연결하여 collision pair를 찾는 전략은 다음과 같습니다. $M[4:8]$이 모두 0x00이고 A=B인 message M을 고려합니다($A_0[4:8] = B_0[4:8] = 0x00$). M에 대한 Hash값 O를 구합니다($O = A_5$). $A1[0:4]$와 같은 값을 만드는 새로운 A_0', B_0'을 만듭니다. $A_0 = B_0$이므로, 같은 위치의 bit를 함께 바꾸면 됩니다. 이렇게 만들어진 M'의 hash값 $O'(= A_5' \oplus A_0' \oplus B_0')$을 계산합니다. 위의 성질들을 통해 첫 4bytes에 대해 $O = O'$입니다. 이제 뒤의 12bytes를 맞춰주기 위해 B_0'의 뒷쪽 12 bytes를 변경해주면 됩니다. $O \oplus (O' \oplus B_0')$의 뒷쪽 12bytes로 B_0'를 변경하게되면, $O = O'$이 됩니다.

2. hash_func.py의 HashFinder class

HashFinder class의 hash method를 통해 과제에서 주어진 hash 함수를 구현했습니다. message M과 hash 과정에서 사용되는 자료형은 Python의 bytearray입니다. F함수의 AES S_BOX는 오픈소스의 array를 가져왔습니다. MixColumn은 times 메서드를 통해 leftshift와 prime polynomial을 통한 xor로 구현했습니다.

문제를 풀기 위한 메서드는 find 메서드 입니다. Random Seed값을 인자로 받아, 이를 기반으로 Random한 Message를 만들고 해당 Message와 collision이 일어나는 다른 Message를 찾습니다. Message를 생성하는 메서드는 makeinput 메서드입니다. 위 글에서 설명했듯, A의 첫 4bytes를 random하게 생성하고, 그 다음 4bytes는 0x00으로 만듭니다. 편의상 첫 4bytes만 random 하게 만들고 나머지를 0x00으로 만든 후, B를 똑같이 만들었습니다. 즉, $M = A||B, A = B$

위에서 만든 M으로 hash값을 생성합니다. 그 후, A와 B의 값을 tweak 메서드를 통해 바꿔주는데, 편의상, A, B의 첫번째 byte의 마지막 bit를 바꿔줬습니다. (xor ^ 0x01) 바꾼 A', B'에 대해 새로운 Hash값 O'을 계산하고, 위에 설명한대로 $O \oplus (O' \oplus B'_0)$로 B'_0의 뒤쪽 4byte를 바꿔주면, 같은 Hash값을 만들어내는 Message $M' = A'||B'$을 찾을 수 있습니다.

main.py에서는 HashFinder 객체를 생성한 후, find 메서드를 통해 seed=17로 설정했을 때, Collision Pair를 찾아 출력합니다. 아래 validity check는 10만개의 서로 다른 예시에 대해 모두 collision pair가 나오는지 check합니다. 저 혼자 체크해보고 주석처리 하였습니다.

3. Collision Pair

```
M1 = 0x85 0x6a 0xce 0x4d 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 # A
     0x85 0x6a 0xce 0x4d 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 # B

M2 = 0x84 0x6a 0xce 0x4d 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 0x0 # A'
     0x84 0x6a 0xce 0x4d 0x6f 0x9f 0x27 0x4d 0xd2 0xf4 0x9a 0xca 0x79 0xf8 0x21 0x1f # B'

Hash = 0x63 0x63 0x63 0x63 0x7c 0xb7 0x14 0xe9 0x54 0x1c 0x63 0xee 0xb7 0x53 0xa9 0x95
```

main.py를 실행했을 때(seed=17) 출력되는 예제 입니다.