PROJ 5

2018037356 안동현

2017012051 백채현

2021029798 바트온드라흐

1. 함수 설명

(1) int rsaes\_oaep\_encrypt(const void \*m, size\_t mLen, const void \*label, const void \*e, const void\* n, void \*c, int sha2\_ndx);

EM을 구성해 공개키로 암호화해줍니다.

먼저 label을 해시해 lHash를 구합니다. 이후

lHash || 0x00…. || 0x01 || m으로 DB를 구성해주고 arc4random\_buf를 통해서 seed값을 얻어냅니다.

해당 seed값을 MGF 해서 dbMask를 얻어냅니다. 이후 dbMask와 DB를 XOR 연산해서 maskedDB를 알아냅니다. 이제 이 maskedDB를 한번 더 MGF해서 seedMask를 얻어내고 seedMask와 seed를 xor연산해줘서 maskedSeed를 얻어냅니다.

이제 EM을 구해줍니다. 가장 첫번째 바이트를 0x00으로 해주고, 이후 maskedSeed, maskedDB 순으로 ||를 해줍니다. EM이 완성되었으므로 공개키 (e, n) 으로 암호화 해줘서 c에 저장해줍니다.

(2) int rsaes\_oaep\_decrypt(void \*m, size\_t \*mLen, const void \*label, const void \*d, const void \* n, const void \*c, int sha2\_ndx);

c를 복호화 해줘서 EM을 구해내고, 이로부터 DB를 구해서 평문 m과 길이 mLen을 복원해냅니다.

먼저 c를 복호화 해줘서 EM을 구해줍니다. 여기서부터 maskedSeed와 maskedDB를 구해주고, maskedDB를 MGF해서 seedMask를 얻어냅니다. 이제 seedMask와 maskedSeed를 XOR연산해주면 seed를 구해줄 수 있습니다. 이제 DB를 구하기 위해 seed를 다시 MGF해주면 dbMask를 구할 수 있고, dbMask와 maskedDB를 XOR연산 해줘서 DB를 알아냅니다.

이제 제대로 DB를 구해냈다면 패딩을 지나 0x01이 나온 이후부터는 평문 m이므로 m을 복원해주고 mLen을 복원해줍니다.

만약 주어진 label값을 해시한 lHash가 DB의 앞부분 즉 해시 부분과 다르다면 데이터가 변조된 것이므로 복호화에 실패합니다.

(3) int rsassa\_pss\_sign(const void \*m, size\_t mLen, const void \*d, const void \*n, void \*s, int sha2\_ndx);

mLen 길이 m을 EM으로 만들어서 개인키(d, n) 으로 서명해서 s에 저장해줍니다.

먼저 m을 해시해서 mHash를 만들어줍니다. 이후 salt값을 알기 위해 arc4random\_buf 를 이용해서 hLen \* sizeof()unsigned char) 길이의 랜덤값을 뽑습니다. 이것이 salt값입니다. 이제 0x00들과 결합해서 m 프라임 즉 M\_P를 만들어냅니다.

이제 M\_P를 해시해서 M\_P\_Hash값을 알아내고, 해당 해시값을 MGF 하여서 dbMask를 구해냅니다. 이제 0x00들과 0x01 그리고 salt값을 합쳐서 DB를 만들어내고,

만들어진 DB와 dbMask를 XOR 연산해서 maskedDB를 구해냅니다.

이제 마지막으로 maskedDB와 M\_P\_Hash, 그리고 0xBC를 || 해서 EM을 구성하고 만약 맨 왼쪽 비트가 1이라면 0으로 만들어줍니다.

이제 이렇게 완성된 EM을 개인키 (d, n) 으로 서명해서 s에 저장해줍니다.

(4) int rsassa\_pss\_verify(const void \*m, size\_t mLen, const void \*e, const void \*n, const void \* s, int sha2\_ndx);

s값을 공개키 (e, n) 으로 검증해줍니다. 검증하는 방법은 s를 공개키를 사용해서 EM으로 만들어 준 후에, EM의 해시값을 얻어냅니다. 그리고 m 프라임 해시값을 알아낸 뒤 두 해시값을 비교해줍니다.

먼저 메시지 m을 해시해 mHash를 얻습니다. 이후 s를 공개키로 풀어 EM을 얻어냅니다.

해당 EM에서 maskedDB와 EM의 해시(EM\_H)를 알아내고 이를 MGF1을 이용해서 dbMask로 바꿔줍니다. 이제 두개를 XOR 연산을 통해 DB값을 알아냅니다. 그런데 해당 DB값은 완벽하지 않습니다. 이전에 sign에서 MSB 비트를 1일 경우 0으로 만들었기 때문입니다. 따라서 DB의 맨 왼쪽 비트는 0이여야 하지만 0이 아닐 수도 있습니다. 이 부분을 참고해야 합니다.

이후 DB 에서 salt값을 뽑아내고 이를 이용해 0x00과 mHash, salt값으로 m 프라임 즉 M\_P를 만들어냅니다. 이제 이것을 해시해주면 M\_P\_Hash값이 나오고, 이것을 EM\_H와 비교해줍니다.

만약 두 해시값이 다르다면 인증 실패입니다.

(5) void MGF1(const unsigned char \*mgfSeed, size\_t seed\_len, size\_t mask\_len, int sha2\_ndx, unsigned char \*T);

seed\_len 길이의 mgfSeed를 ceil((double)mask\_len/hLen) \* hLen 길이만큼으로 바꾸어 T에 저장해주는 함수입니다. 먼저 해시 길이인 hLen을 구해주고, ceil((double)mask\_len/hLen) 만큼 반복문을 돌아서 I2OSP와 매 반복의 i를 이용해 C를 구하고, mgfSeed || C 의 값을 tem 에 넣어줘 해시시켜 digest에 저장해주고 이 T = T || digest를 이용해 T를 완성시켜 나갑니다. 이때 저희는 64비트 정수에서만 한번에 ||가 가능하므로 여기서의 || 는 실제로는 메모리를 확인해가며 배열로 합쳐 저장해줍니다. 이때 주의할 점은 T의 길이는 mask\_len보다 길수도 있고 같을수도 있습니다.

ceil((double)mask\_len/hLen) \* hLen >= mask\_len 이기 때문입니다.

(6) void I2OSP(size\_t x, unsigned char \*t);

MGF1을 위해서 x라는 수를 32비트 정수로 바꾸고 4 단위로 나누어서 각각의 8비트 원소로 나눕니다. 이후 t[0], t[1], t[2], t[3]에 각각의 값을 넣어주고 리턴해줍니다.

(7) int countBits(size\_t num);

주어진 메시지의 길이가 해시를 할 수 있는 크기인지 확인하기 위해 비트의 개수를 새어주는 함수입니다.

(8) static void sha\_gen(int sha2\_ndx, const unsigned char \*message, unsigned int len, unsigned char \*digest);

sha2\_ndx 의 값에 따라서 함수포인터 배열에 각각의 해시 함수들을 담아놓고, 적절한 해시함수를 뽑아서 길이 len, 메시지 message인 데이터를 해시해, digest에 넣습니다.

(9) static int sha\_len(int sha2\_ndx);

sha2\_ndx 의 값에 따라서 미리 정의 되어있는 해시 길이를 리턴합니다.

2. 실행 결과

(1) OAEP

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

먼저 rsa키가 생성되는 모습입니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

첫번쨰 OAEP 예제에 대해서 문제 없이 통과합니다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

나머지 OAEP 문제들 역시 문제 없이 통과가 되고 있고 윤동주 예제가 맞는 것을 보아서 표준 방식을 준수했음을 알 수 있습니다.

(2) PSS

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

PSS예제들에 대해서도 문제 없이 통과가 되고 있습니다. 에러코드들을 보아하니, 예상 출력들과도 일치합니다.

3. 소감

2018037356 안동현

먼저 가장 어려웠던 부분은 MGF에서 배열의 길이를 신경써주는 부분과, ceil(a/b) 를 해줄 때 a/b 에서 (double) 로 캐스팅해주지 않아 오랜 시간이 걸렸던 점이 있습니다. 또한 PSS에서는 OAEP와는 다르게 맨 왼쪽이 0x00인 바이트가 아니라, 0인 비트인 점을 놓치고 있었어서 에러 코드를 잡는데 시간이 조금 걸렸습니다. 또한 적재적소에 에러 메시지를 리턴하는 부분이 보고서를 해석하는 데 시간이 뙈 걸렸습니다. PSS 검증 함수와 MGF, I2OSP, countBits 함수들을 직접 구현하고 팀원들이 어려워 하는 부분들을 도와주고, 전체적인 디버깅을 하면서 모든 함수에 대해 깊이 이해할 수 있어서 좋은 경험이었습니다.

2017012051 백채현

RSASSA-PSS 기법 자체가 RSAES-OAEP 기법과 크게 다르지 않아서 쉽게 구현할 수 있었다. 각 블록들의 길이를 알고 원하는 해쉬 함수로 해쉬를 만들어 주었더니 한번에 테스트를 통과했다. 블록들을 이어붙이는 과정 또한 memset, memcpy 같은 메모리 계열 함수들을 이용해 깔끔하게 만들 수 있었다. 오류도 아예 블록들의 길이를 정해놓는 배열을 사용하여 피해갈 수 있었다. 중간에 만든 MGF 함수에 결함이 있어 다른 팀원이 만든 함수를 사용하였지만, 덕분에 MGF함수의 개념과 원리에 대해 잘 이해할 수 있었다.

2021029798 바트온드라흐

이 프로젝트를 통해 메모리 기능과 함수 호출 방법에 대한 이해를 높일 수 있었습니다. 처음에는 RSAES-oaep 함수를 작성하는 데 어려움이 있었지만, 인터넷에서 조사하고 도움을 구하여 코드를 작성하는 방법을 익힐 수 있었습니다. memcpy 함수는 처음 사용해 보았지만, 이 프로젝트를 통해 그 사용법을 익힐 수 있었습니다. 제 코드에서 일부 문제를 해결했지만 해시 함수와 MGF를 올바르게 처리하는 데 어려움을 겪었습니다. 다행히도 팀원들의 도움으로 이 문제를 극복하고 코드를 수정할 수 있었습니다. 이 프로젝트를 통해 코딩 지식이 확장되었고, 강의에서 배운 OAES에 대한 개념을 실제로 응용하여 암호화와 복호화에 대한 이해가 높아졌습니다.