

Computer Security

Term Project

- Programming a secure online bank -

| | | |
|---|-----|------------|
| 학 | 과 | 컴퓨터공학과 |
| 학 | 번 | 201211704 |
| 이 | 름 | 김기홍 |
| 제 | 출 일 | 2016.12.04 |

<목 차>

| | |
|------------------------------------|----|
| 1 서론 | 2 |
| 2 배경 | 4 |
| 3 시스템 구조 및 시연 | 5 |
| 3.1 최초 시스템 구조 | 5 |
| 3.2 최종 시스템 구조 | 5 |
| 3.3 프로그램 기능 시연 | 6 |
| 4 암호화 모듈 | 12 |
| 4.1 A5/1 | 13 |
| 4.2 Feistel 구조 기반 Block 암호 | 15 |
| 4.3 RSA 암호 | 17 |
| 4.4 ECC 디피-헬먼 키 교환 | 19 |
| 4.5 SHA-256 | 21 |
| 5 상호 인증, 세션키, PFS를 만족하는 프로토콜 | 23 |
| 5.1 보완된 RSA 알고리즘 | 27 |
| 6 트루디의 공격 시연 | 29 |
| 6.1 재전송 공격 테스트 | 29 |
| 6.2 재전송 공격 방지 테스트 | 33 |
| 6.3 중간자 공격 테스트 | 37 |
| 7 회고 | 39 |
| 8 부록 | 45 |

1. 서론

본 프로젝트는 '온라인 뱅킹 시스템'을 기본 모델로 인증, 인허, 암호, 보안 프로토콜 등의 전체적인 보안 시스템을 이해하고 개발하는 것이 목적이다.

최초 프로젝트에서는 은행 서버와 고객 클라이언트로 구성된 2-Tier 구조를 사용했다. 구현했던 기능은 계정 등록, 인증 후 계좌 조회, 입금, 출금 기능과 RSA 알고리즘을 통한 암호화 메시지 전송 기능이었다. 공개키는 각 프로그램 실행 시 공개키, 개인키를 생성한 후 공개키를 서로 교환함으로써 배포하였고 서버는 멀티스레드 방식으로 여러 클라이언트를 동시에 접속시킬 수 있었다. 각 트랜잭션은 모두 파일로 기록되며 인증 정보(ID, PW)는 서버에 파일로 저장되는 방식이었다. 이 시스템의 문제로는 크게 메시지 무결성 문제, 서버 혹은 사용자 인증 문제, 메시지 인증 문제, 서버 측 사용자 정보의 보안 미흡 등이 있었다.

중간 프로젝트에서는 외부에서 중간자가 데이터를 훔쳐보거나 변조가 가능하도록 2-Tier 구조를 3-Tier 구조로 변경했다. 이 과정에서 RSA 암호화를 포함한 스트림 암호 A5/1, Feistel 구조를 사용하여 만든 별도의 블록 암호 모듈을 추가했다. 또한 대칭키 암호의 문제점인 키 교환 문제를 개선하기 위해 ECC 디피-헬먼 방식을 적용한 'Key_Gender' 모듈을 직접 구현하여 사용했다. 이 외에도 서버 측 인증 정보가 노출될 가능성을 고려하여 서버 측 사용자 인증 정보를 암호화했다. 특히 ID, PW는 KISA에서 제공하는 SHA-256 라이브러리를 통해 해시 과정을 거친 후 보관하도록 구현했다. 또한 TCP 프로토콜의 메시지 경계 문제를 고정 길이 메시지를 주고받음으로써 해결하였다. 이전보다 개선시킨 시스템임에도 불구하고 아직 여러 보안적 문제점들이 존재한다. 그 문제점들을 간단하게 설명하자면 우선 이번 시스템은 사용자는 인증하고 있지만 더욱 중요시 되어야 할 서비스를 제공하는 서버를 인증하고 있지 않다는 점이다. 또 이번 시스템은 서버가 사용자의 패스워드를 수신 받아서 해시하기 때문에 사용자들의 패스워드를 알고 있다는 점이다. 단언컨대 패스워드가 알려지는 것은 결코 바람직하지 못한 상황이다. 중간자 공격(Man In The Middle Attack) 및 재전송 공격(Replay Attack) 또한 여전히 유효하며 이는 6장에서 재전송 공격에 대해 실제로 테스트 해본다. 그리고 아직까지 적용되지 않은 보안 프로토콜과 소프트웨어 측면의 위험성 배제 또한 이번 시스템의 풀어야 할 과제이다.

최종 프로젝트에서는 이전까지 구축되었던 시스템에서 추가로 상호 인증, 세션키, PFS(Perfect Forward Secrecy)를 만족하는 프로토콜을 적용하였다. 본 보고서에서는 이 프로토콜을 'PFS 프로토콜'이라는 이름으로 소개한다. PFS 프로토콜을 구현하는 과정에서 기존의 RSA 암호화 알고리즘에 메시지가 일정 수준 이상 커지면 제대로 동작하지 않는다는 문제점을 발견하여 무한 자리수를 다루는 BigInteger 라이브러리와 직접 구현한 간단한 Padding 알고리즘을 적용시켜 해결하였다. 최종 프로젝트에서는 이전까지 문제점이 되었던 단방향 인증 문제를 해결하고, 중간자 공격 및 재전송 공격을 방지하고 비밀키가 노출되더라도

다른 세션키로 암호화된 내용은 해독할 수 없게 시스템이 구축되었다. 또한 서버도 사용자의 패스워드를 알 수 없도록 해시된 NONCE 기법을 사용해 패스워드를 비교한다. 하지만 아직까지도 문제점은 존재한다. 소프트웨어에 대한 취약점이 대표적이다. 이 부분은 최종 시스템에 적용되지는 않지만 추후에 학습을 통해 어떤 취약점이 존재했으며 어떤 대책이 있었는지 회상함으로써 마치기로 한다.

이 보고서에서는 2장에서 먼저 프로젝트를 진행한 시스템의 사양을 소개하고 진행된 프로젝트를 순차적으로 소개한다. 3장에서는 진행에 따른 프로젝트의 시스템 구조, 4장에서 암호화 모듈의 구현 및 동작을 설명하고 5장에서는 최종 프로젝트에서 추가된 상호 인증, 세션키, PFS를 모두 만족하는 프로토콜에 대해 설명한다. 6장에서 중간자 공격 및 재전송 공격에 대한 대책이 세워지기 전 시스템에 대해 트루디가 중간 서버로 침입하여 데이터를 Cut and Paste하는 재전송 공격을 보이고 이 공격을 방지할 수 있도록 시스템을 보완한다. 더 나아가 중간자 공격 테스트도 보인다. 그리고 7장에서 시스템이 보완되어가는 과정에 대해 설명하고 진행 과정에서의 문제점과 해결 과정 및 배운 점을 기술한다. 마지막 8장 부록에서는 고전 암호 중 카이사르 암호를 소개 및 시연하고 카이사르 암호문을 적용시킨 에드거 앨런 포의 소설 『The Gold Bug』의 알파벳 빈도수를 조사하여 키 전수 검사 없이 한 번에 Breaking하는 프로그램을 소개한다.

2. 배경

우선 이 프로젝트를 진행한 컴퓨터의 시스템 정보는 다음과 같다.

| 구분 | 내용 |
|-------|---|
| 운영 체제 | Microsoft Windows 10 Pro (64-bit) |
| 프로세서 | Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz 2.40 GHz |
| 메모리 | 8.00GB |

그림 2.1 프로젝트 환경

그리고 다음은 사용된 툴들의 정보이다.

| 사용 목적 | 툴 이름 |
|-------|--|
| 개발 | Microsoft Visual Studio Community 2015 |
| 분석 | Wireshark 2.2.0 (64-bit), RawCap |

그림 2.2 사용한 툴

해당 툴들을 선택한 이유는 작성자가 기본적으로 사용하는 툴들이고, 그만큼 조작에 있어서 다른 툴들에 비해 비교적 능숙하기 때문이다.

‘온라인 뱅킹’ 시스템의 구조는 최초 TCP 연결 기반의 서버, 클라이언트로 이루어졌지만, 중간에서 패킷 내용을 변조하기 위해서 다음과 같이 조정하였다. 새로 조정된 시스템 구조는 TCP 연결 기반의 서버, 릴레이 서버, 클라이언트로 이루어지고, 서버와 릴레이 서버는 멀티 스레드 환경을 지원하기 때문에 여러 클라이언트가 동시에 접속할 수 있다. 릴레이 서버의 포트 번호는 9000번, 서버의 포트 번호는 9001번을 사용하였다.

3. 시스템 구조 및 기능 시연

3.1. 최초 시스템 구조

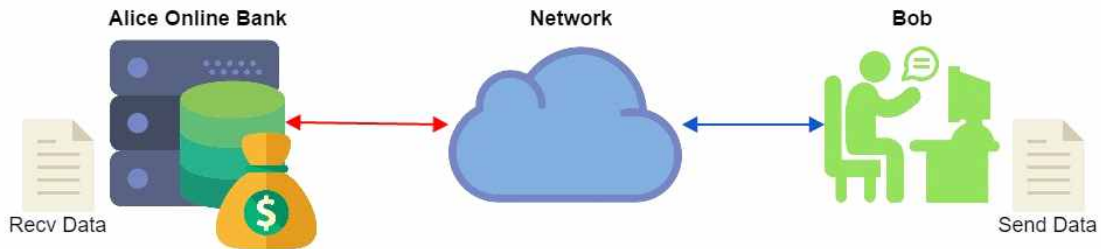


그림 3.1.1 최초 시스템 구조

그림 3.1.1은 최초 프로젝트의 시스템 구조를 보인다. 서버와 클라이언트 1:1 관계를 가지며 통신한다. 이 구조에서는 지나가는 패킷을 가로채어 변조하는 테스트가 어려우므로 3.2절과 같이 개선되었다.

3.2. 최종 시스템 구조

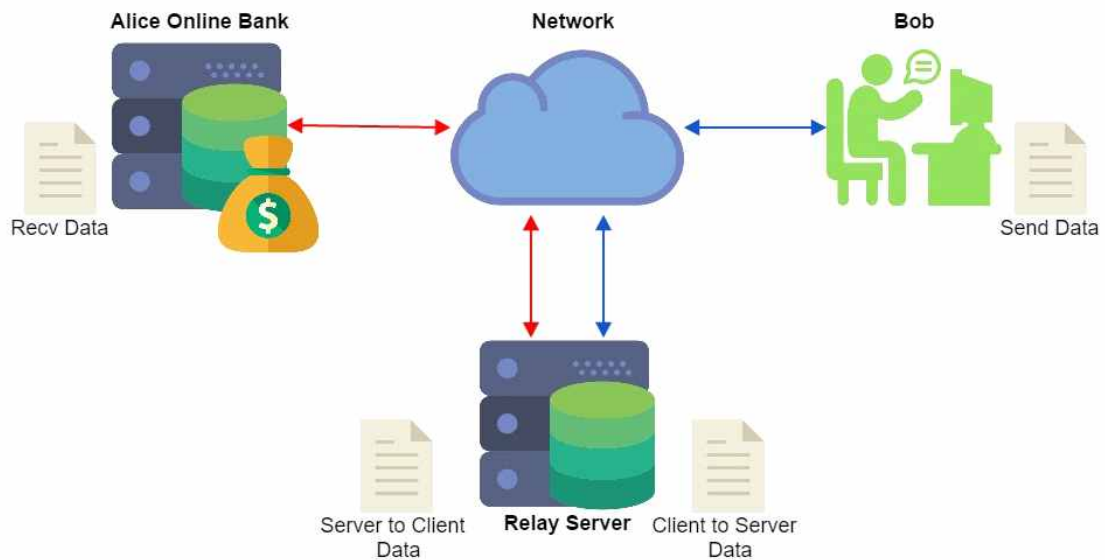


그림 3.2.1 최종 시스템 구조

그림 3.2.1는 최종 시스템의 전체적인 모습을 보여주는 다이어그램이다. 최초 클라이언트 프로그램 실행 시 릴레이 서버의 한 스레드로 접속하고, 그 스레드는 다시 서버의 한 스레드로 접속한다. 따라서 모든 메시지는 릴레이 서버를 거친다.

3.3. 프로그램 기능 시연

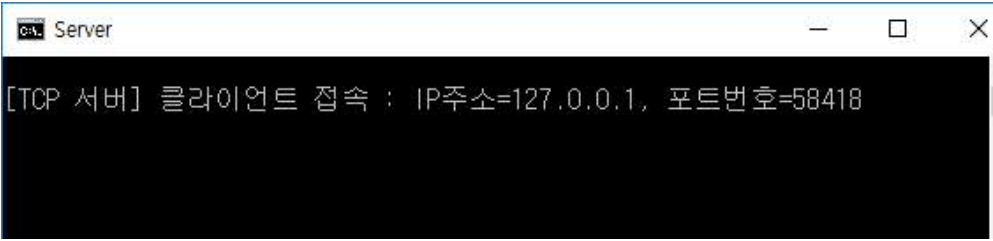
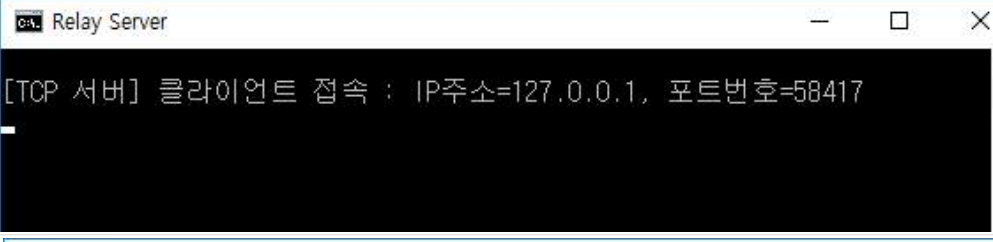

| | |
|--------|--|
| 서버 |  |
| 릴레이 서버 |  |
| 클라이언트 |  |

그림 3.3.1 프로그램 실행 시 화면

그림 3.3.1은 각 프로그램들을 실행하고 연결이 수립되었을 때의 화면이다. 각 프로그램은 각자 파일 입출력을 통해 어떤 정보를 관리한다. 예를 들어, 클라이언트는 수행한 트랜잭션의 종류 및 전송한 정보를 로그 파일 형식으로 저장하고, 릴레이 서버는 클라이언트에서 서버로, 또 서버에서 클라이언트로 전송되는 정보들을 저장한다. 마지막으로 서버는 사용자 정보 즉, ID와 PW, 이름, 잔액을 저장한다.

| | |
|--------|--|
| 서버 |  |
| 릴레이 서버 |  |
| 클라이언트 |  |

그림 3.3.2 계정을 등록하는 화면

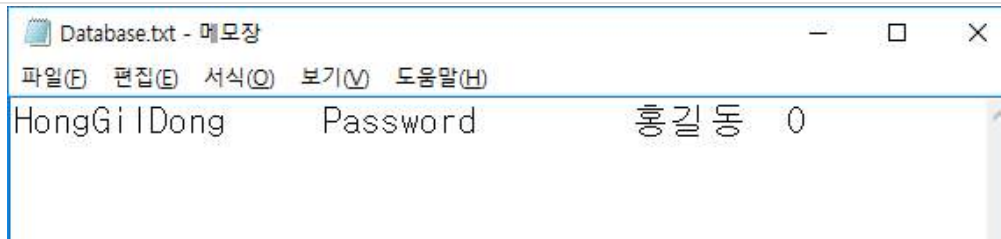
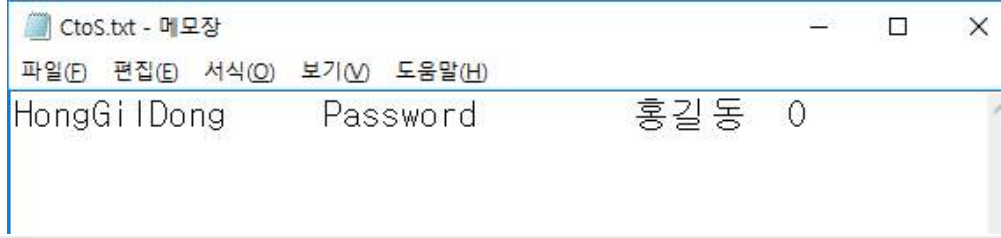
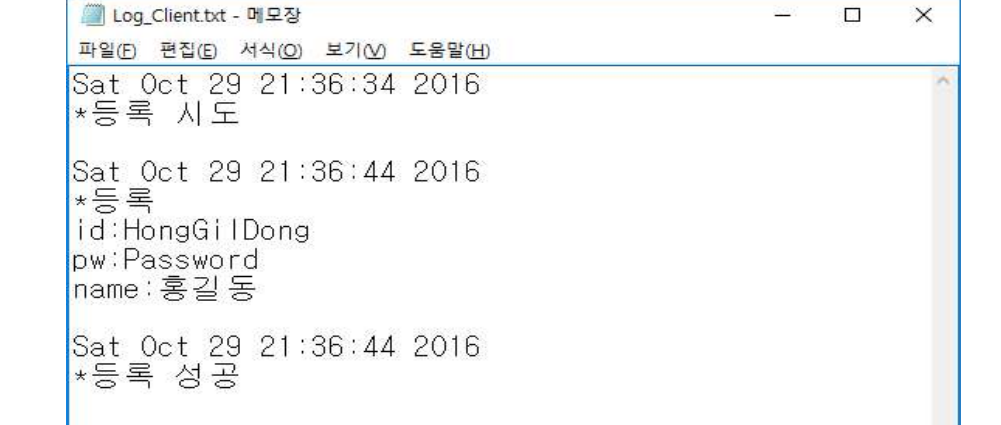
| | |
|--------|--|
| 서버 |  |
| 릴레이 서버 |  |
| 클라이언트 |  |

그림 3.3.3 각 프로그램이 생성한 파일_1

릴레이 서버는 클라이언트가 전송한 데이터를 그림 3.3.3에서 보이는 바와 같이 파일로 저장한다. 그리고 다시 그 파일을 읽어서 서버에게 전송한다. 이는 트루디가 전송되는 내용을 변조할 수 있도록 한 것이다.

| | |
|--------|---|
| 서버 |  |
| 릴레이 서버 |  |
| 클라이언트 |  |

그림 3.3.4 로그인하는 화면

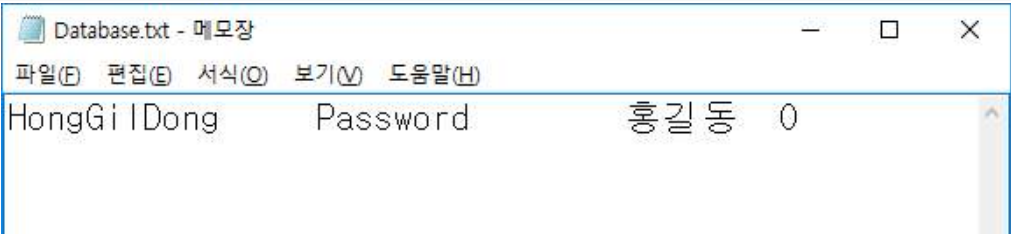
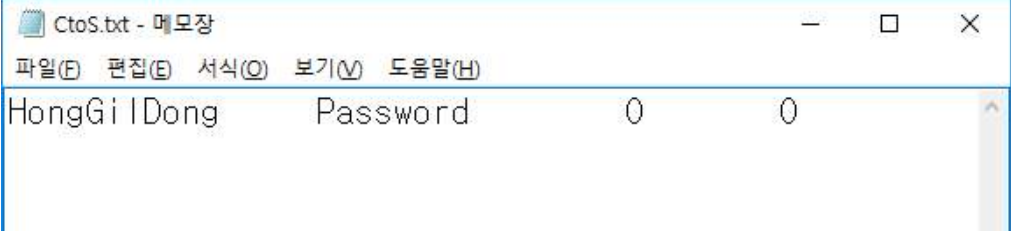
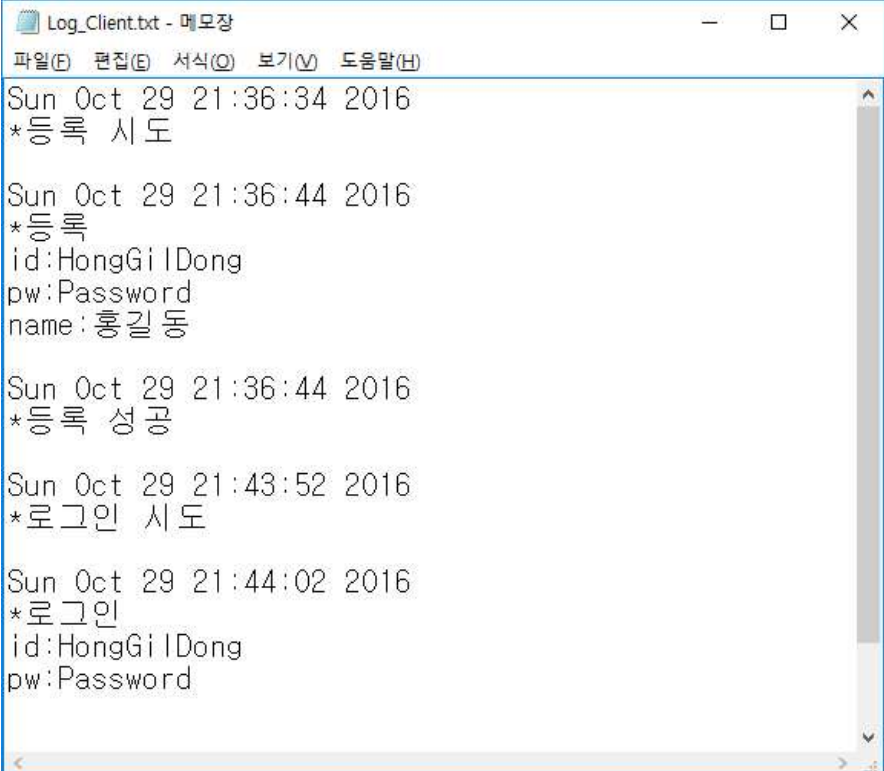
| | |
|--------|---|
| 서버 |  <pre> Database.txt - 메모장 파일(F) 편집(E) 서식(O) 보기(V) 도움말(H) HongGiIDong Password 홍길동 0 </pre> |
| 릴레이 서버 |  <pre> CtoS.txt - 메모장 파일(F) 편집(E) 서식(O) 보기(V) 도움말(H) HongGiIDong Password 0 0 </pre> |
| 클라이언트 |  <pre> Log_Client.txt - 메모장 파일(F) 편집(E) 서식(O) 보기(V) 도움말(H) Sun Oct 29 21:36:34 2016 *등록 시도 Sun Oct 29 21:36:44 2016 *등록 id:HongGiIDong pw>Password name:홍길동 Sun Oct 29 21:36:44 2016 *등록 성공 Sun Oct 29 21:43:52 2016 *로그인 시도 Sun Oct 29 21:44:02 2016 *로그인 id:HongGiIDong pw>Password </pre> |

그림 3.3.5 각 프로그램이 생성한 파일_2

서버는 실행 시, 자신이 보관하고 있는 Database.txt 파일을 읽어서 그 정보들을 연결 리스트에 보관한다. 실행 중 클라이언트가 계정을 등록할 경우 아이디를 비교하여 중복이 없다면 연결리스트에 추가하고, 그 정보를 파일에도 쓴다. 클라이언트가 로그인을 시도할 경우 전송받은 정보와 연결리스트의 정보를 비교하여 인증 과정을 수행한다.



그림 3.3.6 계좌 조회, 입금, 출금 화면 및 Database.txt 내용 변화

계좌 조회, 입금, 출금 기능이 수행될 때마다 서버는 전송받은 값이 타당한지 검사한 후에 자신의 연결리스트 및 Database.txt를 갱신하고, 클라이언트에게 이름 및 잔액을 전송한다.

일반적인 기능 소개는 마치고 오류가 발생할 수 있는 다양한 경우의 화면들을 한번 살펴보고 이번 장을 마치도록 한다.

| | |
|---------------|--|
| 중복된 아이디 등록 |  |
| 아이디 불일치 |  |
| 비밀번호 불일치 |  |
| 잘못된 금액 입금 |  |
| 잘못된 금액 출금 |  |
| 잘못된 메뉴 선택 |  |

그림 3.3.7 다양한 오류 화면

4. 암호화 모듈



그림 4.1 암호화 모듈 및 공통 모듈 라이브러리

암호화 모듈들은 서버, 릴레이 서버, 클라이언트 프로그램에 공통으로 사용되는 모듈들과 함께 하나의 라이브러리로 구성되어진다. 이번 장에서는 구체적인 구현 설명은 뒤로하고, 사용하는 방법과 실제 동작 화면을 보인다.

4.1. A5/1

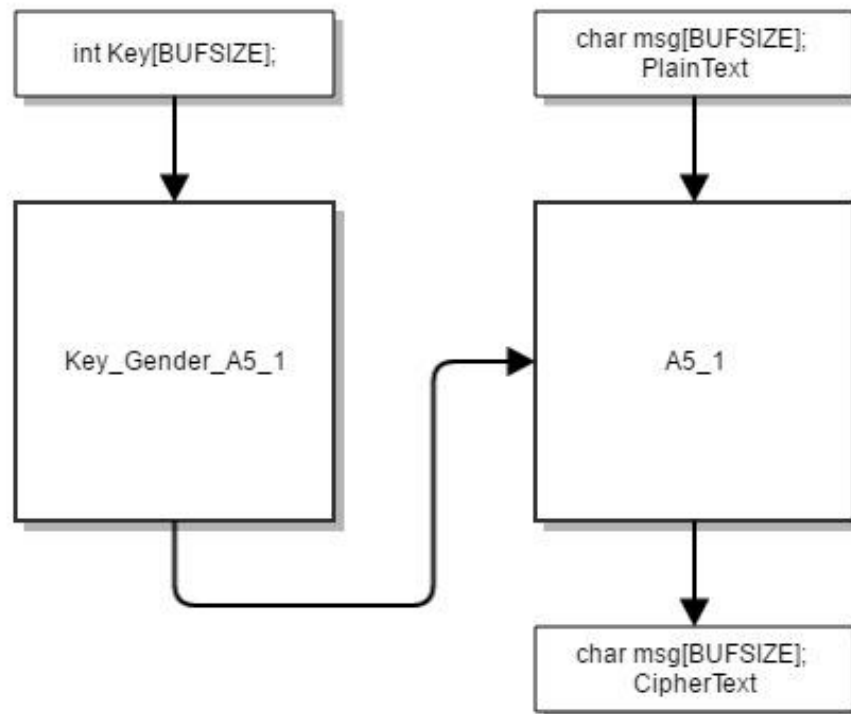


그림 4.1.1 A5/1 암호화 모듈 사용법

A5/1 암호화는 그림 4.1.1처럼 정수형 배열을 선언한 후, Key_Gender_A5_1 함수를 사용해 각 인덱스 하나 당 8비트짜리 키스트림을 연달아 생성한다.

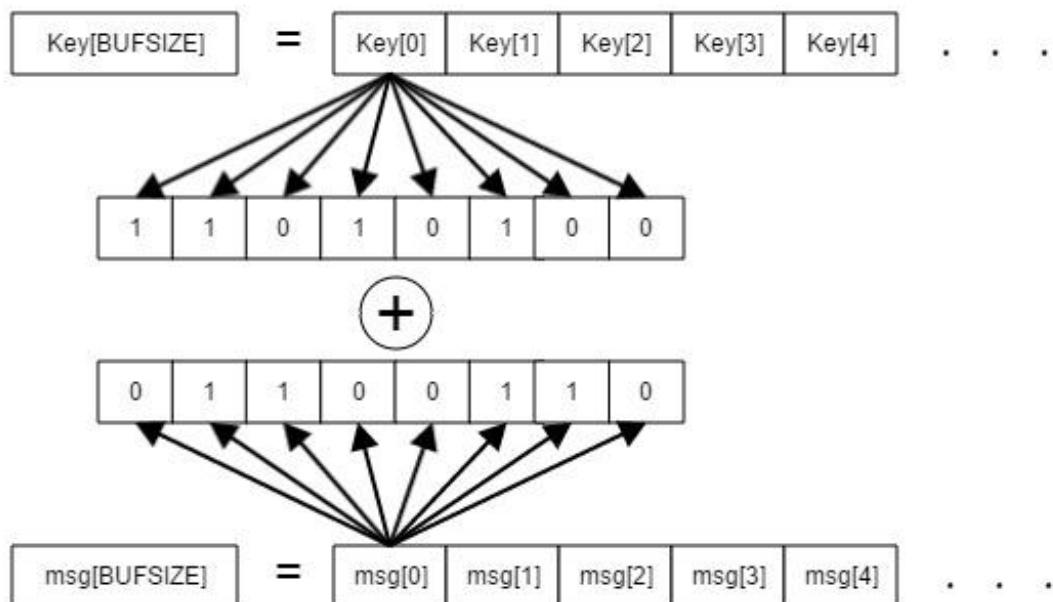


그림 4.1.2 A5/1 키스트림 구조

암호화는 그림 4.1.2에 나와 있듯이 문자열의 각 문자 하나당 이루어진다. 8비트짜리 키스트림을 저장하는 배열은 int형이 아닌 short형으로 구현하는 것이 메모리 효율에 좋지만 작성자 판단에 여러 암호화 모듈에서 사용할 수 있도록 정수형을 선택해 작성하였다.

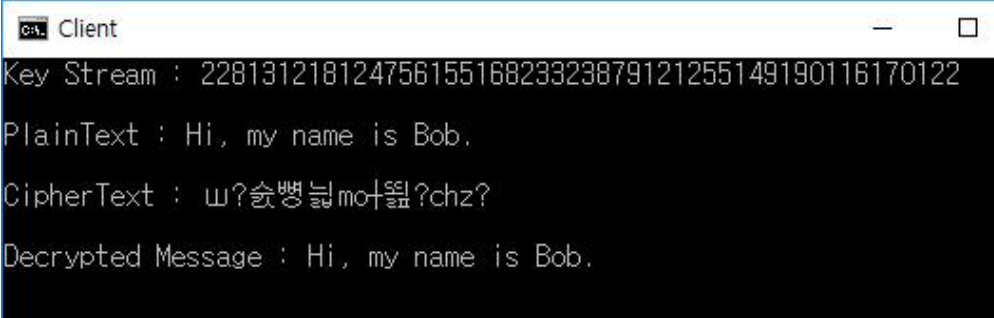
| | |
|----------------------------------|--|
| <p>A5/1 암호화 시연</p> |  |
| <p>A5/1 암호화 시연 코드</p> | <pre>int Key[BUFSIZE]; Key_Gender_A5_1(Key); printf("Key Stream : "); for (int i = 0; i < BUFSIZE; i++) printf("%d", Key[i]); puts(""); puts(""); strcpy(msg, "Hi, my name is Bob."); // 평문 printf("PlainText : %s WnWn", msg); A5_1(msg, Key); // 암호화 printf("CipherText : %s WnWn", msg); A5_1(msg, Key); // 복호화 printf("Decrypted Message : %s WnWn", msg);</pre> |

그림 4.1.3 A5/1 암호화 시연 화면

4.2. Feistel 구조 기반 Block 암호

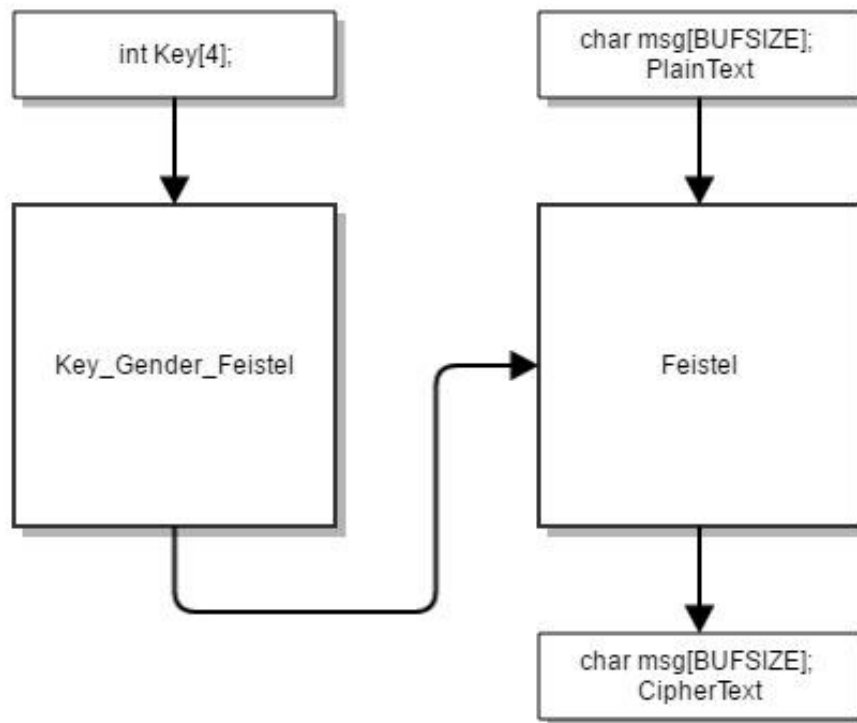


그림 4.2.1 Feistel 구조 블록 암호화 모듈 사용법

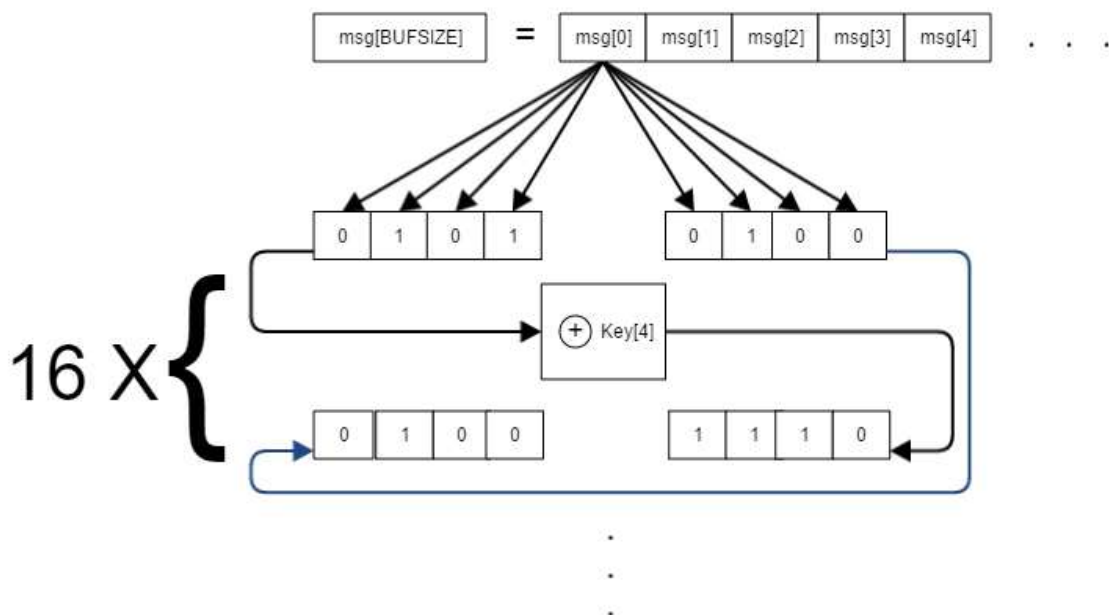


그림 4.2.2 Feistel 구조 블록 암호화 알고리즘

Feistel 구조 암호화는 정말 단순한 알고리즘을 사용하였다. 그 이유는 우선 블록 암호의 구조 중 하나인 Feistel 구조의 이해와 구현에 우선 중점을 두었기 때문이다. 문자 하나를 하나의 블록으로 정하여, 좌우 4비트씩 나눈다. 그리고 오른쪽 블록은

그대로 다음 왼쪽 블록으로, 왼쪽 블록은 4비트 키와 XOR연산을 한 후 오른쪽 블록으로 보낸다. 이 과정을 16번 반복하는 알고리즘이다. 추후에는 블록 암호의 모드를 구현함으로써 메시지 무결성을 보장하는 데에도 기여를 할 것으로 예상된다. 키가 4비트이기 때문에 보안적인 측면에서는 정말 극단적으로 취약하지만, 오히려 구조를 이해하고 직접 구현하는 데는 큰 도움이 되었다.

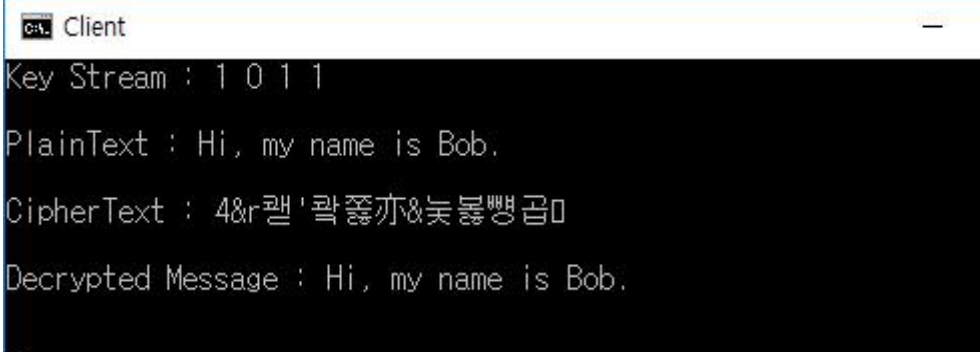
| | |
|--|--|
| <p>Feistel 구조 블록 암호화 시연</p> |  <pre> Client Key Stream : 1 0 1 1 PlainText : Hi, my name is Bob. CipherText : 4&r광'광 좋亦&늣뵡뵡곰 Decrypted Message : Hi, my name is Bob. </pre> |
| <p>Feistel 구조 블록 암호화 시연 코드</p> | <pre> int Key[BUFSIZE]; Key_Gender_Feistel(Key); printf("Key Stream : "); for (int i = 0; i < 4; i++) printf("%d ", Key[i]); puts(""); puts(""); strcpy(msg, "Hi, my name is Bob."); // 평문 printf("PlainText : %s WnWn", msg); Feistel(msg, Key, ENC); // 암호화 printf("CipherText : %s WnWn", msg); Feistel(msg, Key, DEC); // 암호화 printf("Decrypted Message : %s WnWn", msg); </pre> |

그림 4.2.3 Feistel 구조 블록 암호화 시연 화면

4.3. RSA 암호

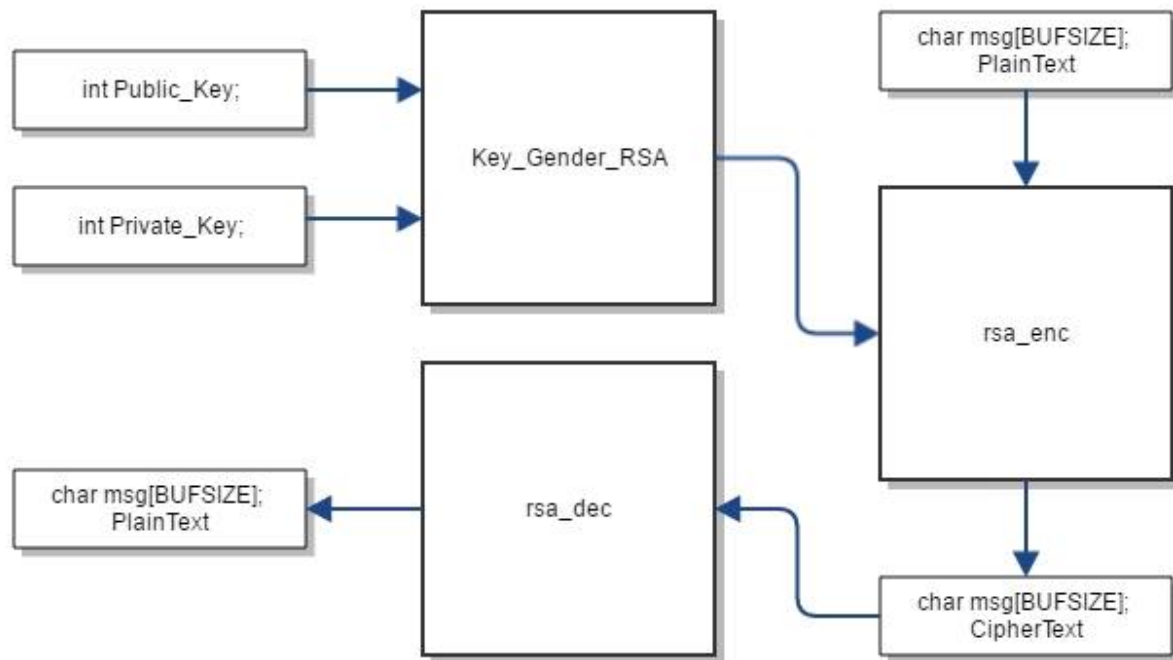


그림 4.3.1 RSA 암호화 모듈 사용법

난수로 만들어진 소수 p , q 와 그 소수들로 구한 e_{pi} 값을 사용해 공개키와 개인키를 `Key_Gender_RSA` 모듈로 구하고, 그 키들을 사용하여 `rsa_enc`, `rsa_dec` 모듈로 문자열을 암호화 및 복호화 할 수 있다. 기존의 RSA 공개키와 개인키는 `int`형을 사용하였지만 추후에 설명할 보완된 RSA 알고리즘에서는 무한 자리수 연산이 가능한 `BigInteger`형을 사용한다.

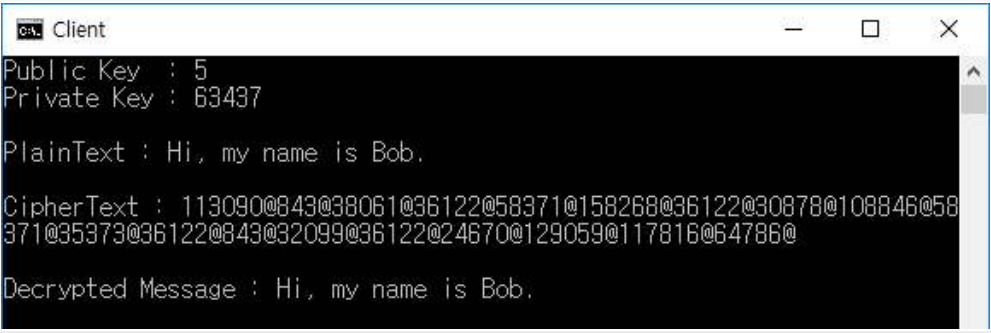
| | |
|---------------------------------|---|
| <p>RSA 암호화 시연</p> |  |
| <p>RSA 암호화 시연 코드</p> | <pre> int public_key, private_key; Key_Gender_RSA(&public_key, &private_key); printf("Public Key : %d\n", public_key); printf("Private Key : %d\n", private_key); puts(""); strcpy(msg, "Hi, my name is Bob."); // 평문 printf("PlainText : %s\n\n", msg); rsa_enc(msg, public_key); // 암호화 printf("CipherText : %s\n\n", msg); rsa_dec(msg, private_key); // 암호화 printf("Decrypted Message : %s\n\n", msg); </pre> |

그림 4.3.2 RSA 암호화 시연 화면

여기서 메시지는 각 문자별로 아스키코드를 사용해 암호화 및 복호화 되는데, 이 때 RSA는 숫자를 사용하기 때문에 각 문자를 구별할 수 있도록 어떤 토큰이 필요하다. 작성자는 각 문자의 암호화된 숫자 사이에 '@' 문자를 삽입함으로써 문자들을 구별할 수 있었다. 예를 들면 다음과 같다.

H i _ B o b
 ~@~@~@~@~@~@

'~'는 각 문자에 해당하는 암호화된 숫자를 의미한다. 위와 같은 모양으로 메시지를 전송하게 되면, 수신측에서는 '@'를 토큰삼아 얻어진 각각의 숫자들을 자신의 개인키로 복호화하게 되면 본래의 메시지를 얻을 수 있다.

4.4. ECC 디피-헬먼 키 교환

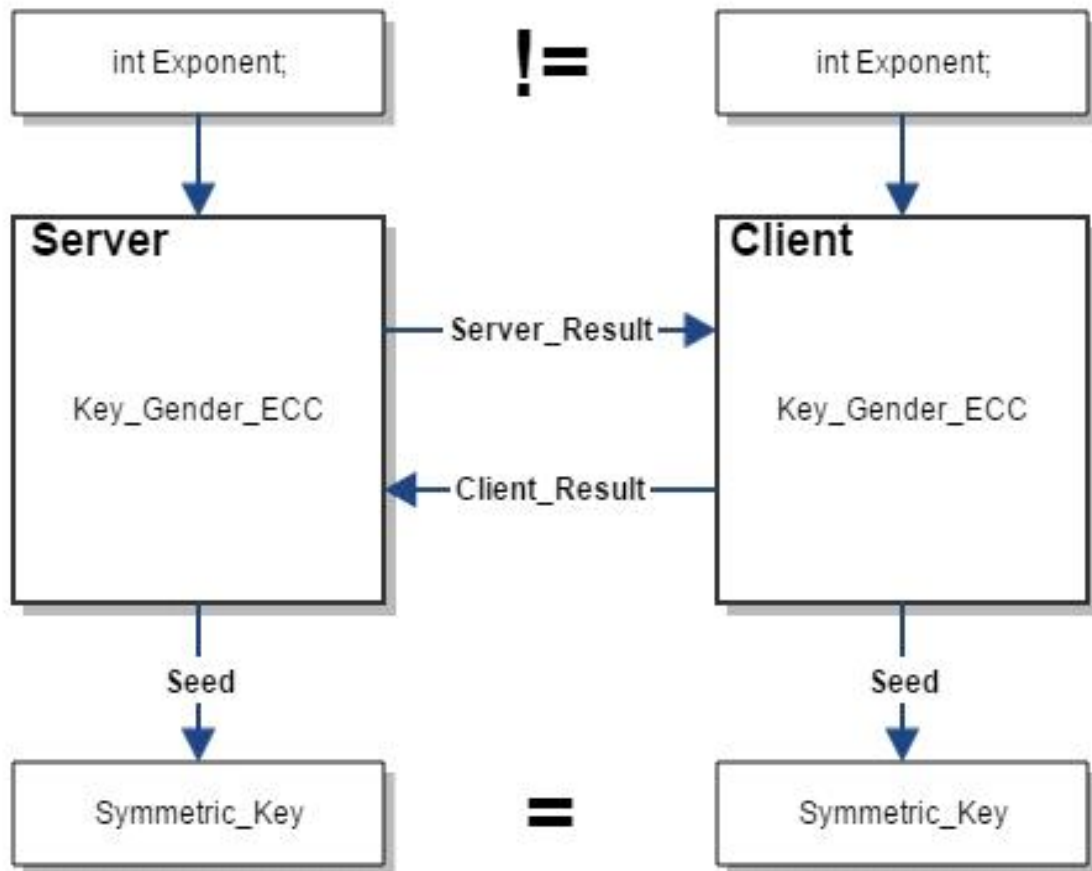


그림 4.4.1 ECC 디피-헬먼 키 교환 모듈 사용법

ECC 디피-헬먼 키 교환 모듈은 각자의 비밀 승수를 모듈에 매개변수로 넣으면 자동적으로 내부에서 ECC 연산이 이루어지고 통신이 이루어진다. 그 후 다시 한 번 더 ECC 연산이 이루어지면 양쪽이 타원 곡선 위의 같은 한 점을 얻게 되는데, 그 점을 Seed로 사용하여 동일한 Symmetric_Key를 생성한다. ECC 연산을 위한 타원 곡선 방정식 및 나머지 연산 값과 개인의 승수 값은 랜덤하게 생성하지만 이번 시연에서는 우선 교과서에 나오는 값으로 설정 해놓았다. 그 이유는 교과서에 나오는 값을 사용함으로써 제대로 된 연산이 이루어지는 것을 증명하기 위해서이다.

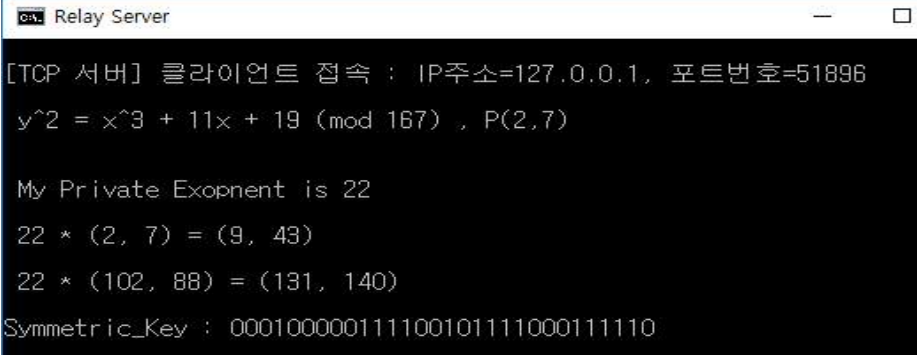
| | |
|--|--|
| <p>서버 ECC 디피-헬먼 키 교환 시연</p> |  <pre> [TOP 서버] 클라이언트 접속 : IP주소=127.0.0.1, 포트번호=51896 y^2 = x^3 + 11x + 19 (mod 167) , P(2,7) My Private Exopnent is 22 22 * (2, 7) = (9, 43) 22 * (102, 88) = (131, 140) Symmetric_Key : 000100000111100101111000111110 </pre> |
| <p>서버 ECC 디피-헬먼 키 교환 시연 코드</p> | <pre> d Thread_Proc(void* sock) SOCKET client_sock = (SOCKET)sock; int Bob_Exponent = 22; int Symmetric_Key[BUFSIZE]; Key_Gender_ECC(client_sock, Symmetric_Key, Bob_Exponent); printf("Symmetric_Key : "); for (int i = 0; i < BUFSIZE; i++) printf("%d", Symmetric_Key[i]); puts(""); </pre> |

그림 4.4.2 서버 측 ECC 디피-헬먼 키 교환 시연 화면

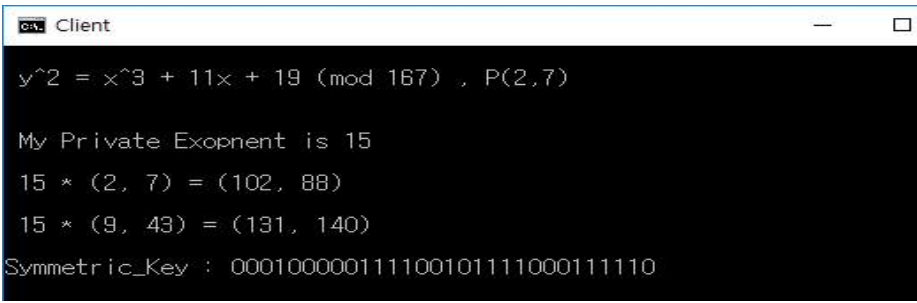
| | |
|---|---|
| <p>클라이언트 ECC 디피-헬먼 키 교환 시연</p> |  <pre> y^2 = x^3 + 11x + 19 (mod 167) , P(2,7) My Private Exopnent is 15 15 * (2, 7) = (102, 88) 15 * (9, 43) = (131, 140) Symmetric_Key : 000100000111100101111000111110 </pre> |
| <p>클라이언트 ECC 디피-헬먼 키 교환 시연 코드</p> | <pre> sock = Make_Client(RELAY_PORT, "127.0.0.1"); int Alice_Exponent = 15; int Symmetric_Key[BUFSIZE]; Key_Gender_ECC(sock, Symmetric_Key, Alice_Exponent); printf("Symmetric_Key : "); for (int i = 0; i < BUFSIZE; i++) printf("%d", Symmetric_Key[i]); puts(""); </pre> |

그림 4.4.3 클라이언트 측 ECC 디피-헬먼 키 교환 시연 화면

4.5. SHA-256

SHA-256 라이브러리는 KISA에서 배포하는 라이브러리를 활용하였으며, 메시지를 넣으면 해시된 메시지를 얻을 수 있는 외부에서 보기에는 간단한 구조이므로 따로 다이어그램으로 소개하지는 않는다. 대신 이번 절에서는 해시 함수의 동작 시연과 더불어 서버 측의 인증 정보 파일에서도 특히 ID와 PW를 해시하여 보관한 모습을 보인다.

해시와 더불어 다른 데이터를 위해 암호화도 병행하는 게 옳지만 우선 여기서는 해시함수의 시연이 주된 주제이므로 암호화는 따로 하지 않는다. 암호화는 앞서 소개한 모듈들을 사용하여 클라이언트, 릴레이 서버, 서버 측의 통신 할 메시지를 전송 시 암호화, 수신 시 복호화 해주면 쉽게 할 수 있기 때문에 우선은 생략한다.


| | |
|-----------------------------|--|
| <p>SHA-256 해시 시연</p> |  <pre> Client Plain Text : HongGilDong Hashed Text : eL헐j'> ?.↔츨꺠꺠p츨?}h_N}9뽕 Plain Text : Password Hashed Text : 靈>涕l9츨0,oa.뽕?+&뽕N+9???→ </pre> |
| <p>SHA-256 해시 시연 코드</p> | <pre> strcpy(msg, "HongGilDong"); printf("Plain Text : %s\n", msg); SHA256_Encrypt((BYTE*)msg, strlen(msg), (BYTE*)log); printf("Hashed Text : %s\n\n", log); memset(log, 0, sizeof(log)); sprintf(msg, "Password"); printf("Plain Text : %s\n", msg); SHA256_Encrypt((BYTE*)msg, strlen(msg), (BYTE*)log); printf("Hashed Text : %s\n\n", log); </pre> |

그림 4.5.1 SHA-256 시연 화면

서버 측 해시된 인증 정보 파일

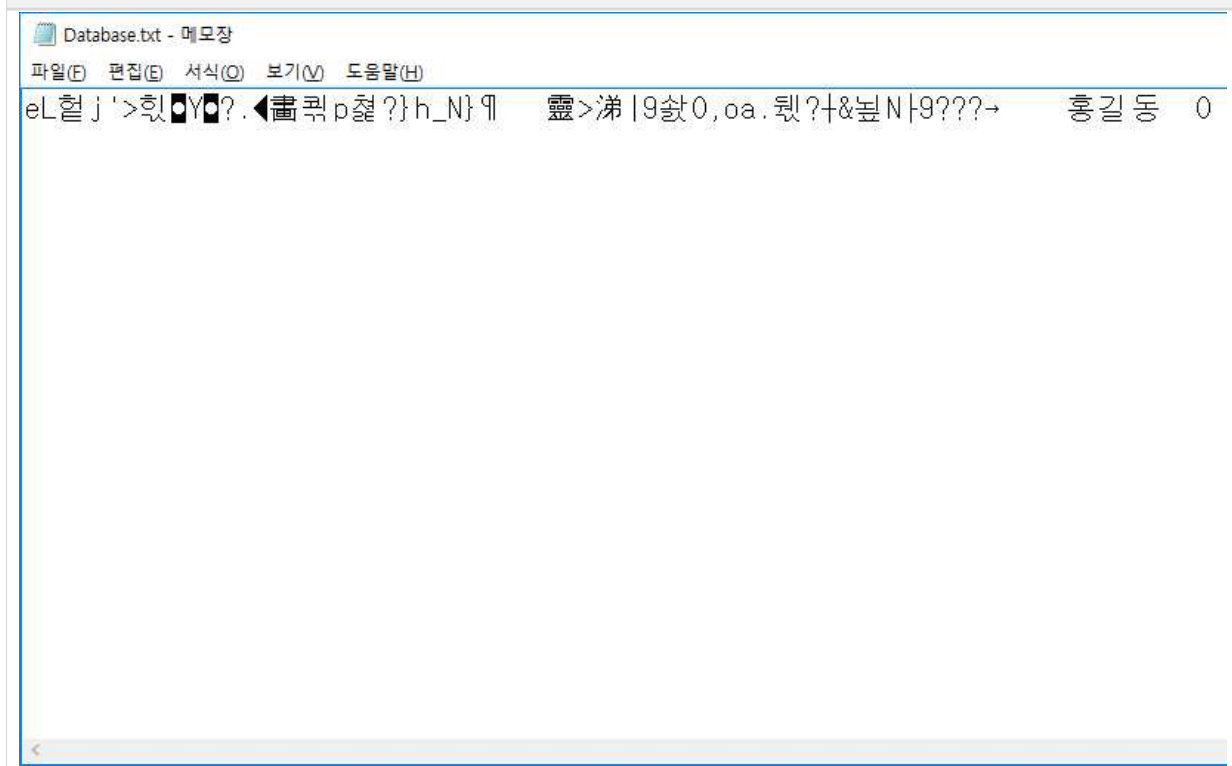


그림 4.5.2 SHA-256 시연 화면(Database.txt)

5. 상호 인증, 세션키, PFS를 만족하는 프로토콜

최종 프로젝트에는 상호 인증과 세션키, PFS를 모두 만족하는 프로토콜(이하 PFS 프로토콜)이 적용되었다. 우선 RSA 알고리즘을 통해 생성된 공개키와 개인키 그리고 두 소수를 곱한 N 을 생성한다. 사전에 서버와 클라이언트는 공개키와 N 을 교환한다. 아래 그림 5.1은 그 이후 프로토콜이 동작하는 과정을 다이어그램으로 나타낸 것이다.

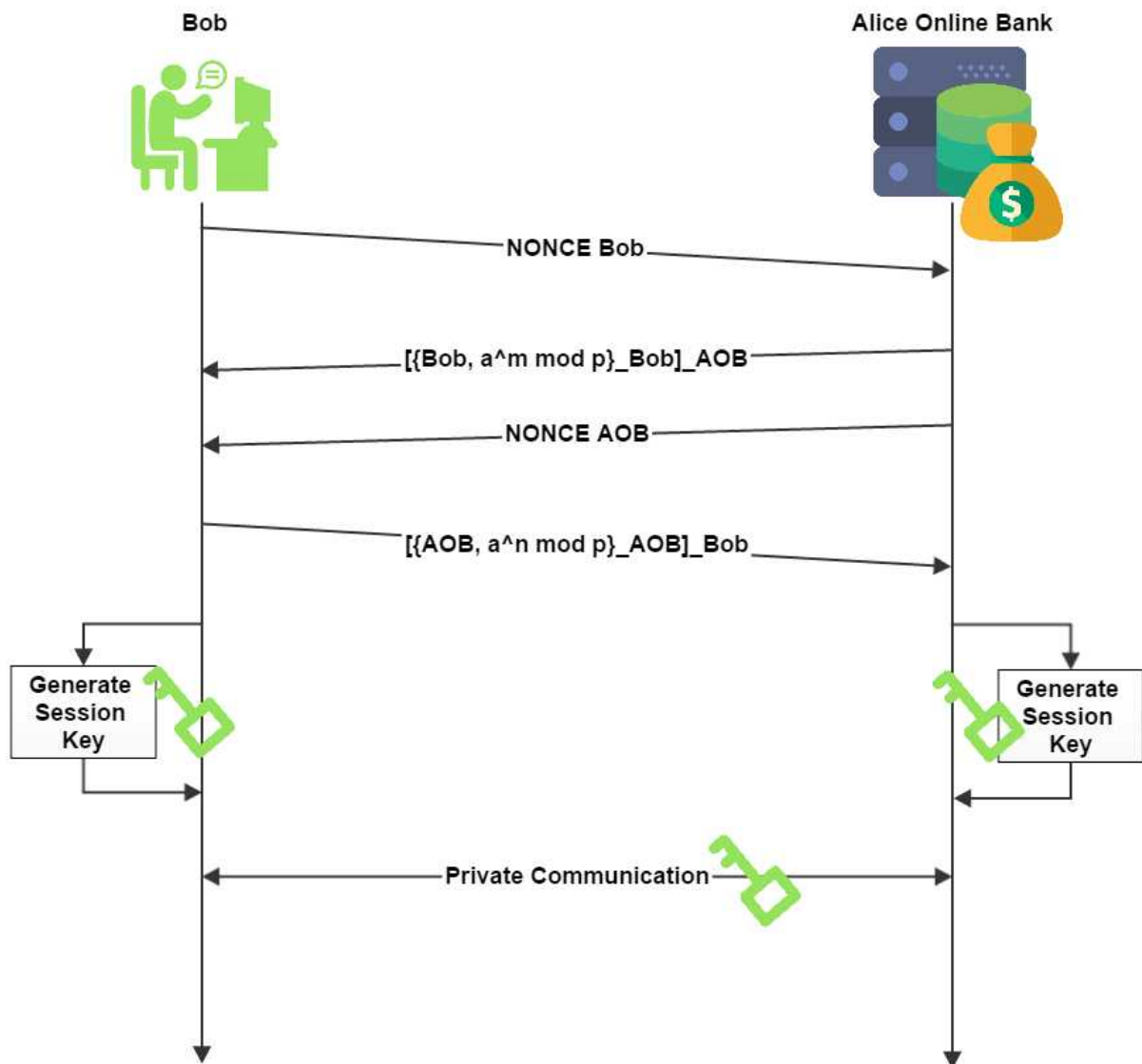


그림 5.1 PFS 프로토콜

Bob은 먼저 난수를 사용한 NONCE를 Alice Online Bank(이하 AOB)에게 전송한다. 이 NONCE는 통신마다 한번 씩만 사용되는 것이 원칙으로 트루디가 재전송 공격을 하는 것을 방지한다. 이후에 AOB는 Bob이 보낸 NONCE와 ECC 디피-헬먼 키 교환 알고리즘을 사용하여 구해진 점을 Bob의 공개키로 암호화하고 이를 다시 자신의 개인키로 서명하여 전송한다. 이 후 자신의 NONCE도 전송한다. Bob 또한 같은 방식으로 서버에게 메시지를 전달한다. 이 과정에서 어떤 메시지를 개인키를 통해 서명했다는 사실과

개인키를 통해 복호화했다는 사실로 서로를 인증할 수 있으므로 중간자 공격 또한 방지한다. 이 후 ECC 디피-헬먼 키 교환 알고리즘을 마침으로써 Bob과 AOB는 서로 같은 비밀키를 안전하게 공유할 수 있게 된다. 설령 트루디가 RSA 개인키 혹은 통신 과정의 세션키를 확보한다고 해도 각각의 세션키는 연관성이 없기 때문에 부분적인 데이터만 해독할 수 있을 뿐이다.

```

Server
[TCP 서버] 클라이언트 접속 : IP주소=127.0.0.1, 포트번호=64062
Public : 395 Private : 161819
Client's Public Key : 721
Client's N : 333943
Nonce_S : 151884
Nonce_C : 335
(2,7) * 8 = (131,140)
평문
335,131,140
암호화
131843 131843 280154 107552 265475 131843 265475 107552 265475 242220 310262
서명
91690 91690 81001 183974 96981 91690 96981 183974 96981 14357 18315
전송용 문자열
91690@91690@81001@183974@96981@91690@96981@183974@96981@14357@18315@
전달 받은 데이터
294148 3163 294148 318881 318881 113432 91646 247451 157689 91646 73917 3163
서명 풀기
143747 68629 143747 21929 21929 14136 85715 92074 125143 85715 169387 68629
복호화
151884,20,35
(20,35) * 8 = (94,12)
94 , 12
계속하려면 아무 키나 누르십시오 . . .

```

그림 5.2 Server PFS 프로토콜 동작 화면

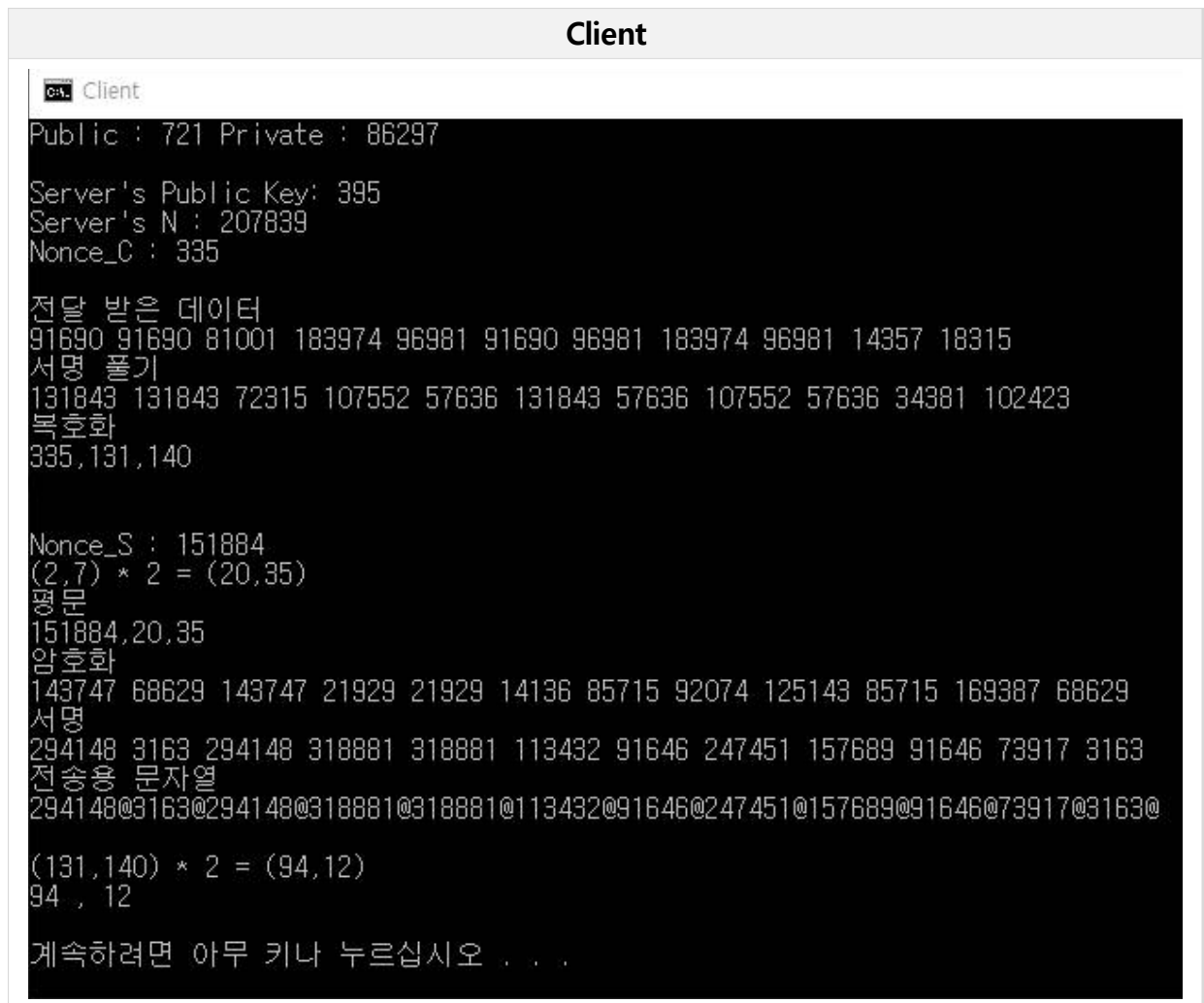


그림 5.3 Client PFS 프로토콜 동작 화면

그림 5.2와 5.3은 PFS 프로토콜을 사용해서 세션키를 만들어내는 화면이다. 각 화면을 보면 평문에 해당하는 글자 하나 당 RSA 암호화를 하는 것을 볼 수 있다. 송신 측에서 암호화와 서명을 마친 문자들은 전송용 문자로 변환되어 전송되고 수신 측에서 다시 각각의 문자로 나누어진 후 반대 순서로 연산을 진행하면서 원래의 평문을 얻을 수 있다. ECC 디피-헬먼 알고리즘에 사용된 곡선 방정식은 교과서에 나온 식과 같으며 공개된 점 역시 (2,7)을 사용하였다. 서버와 클라이언트는 최종으로 나온 점(94, 12)을 Seed 값으로 하여 rand() 함수를 사용하면 안전하게 공유된 세션키를 가질 수 있다.

WireShark를 통한 PFS 프로토콜

| | | | | | | | | | |
|------|-----|------------|------------|----------|----------|------------|---------|----|-------------------------|
| TCP | 552 | 64062→9001 | [PSH, ACK] | Seq=1 | Ack=1 | Win=525568 | Len=512 | | |
| 0020 | 50 | 18 | 08 | 05 | b2 | e6 | 00 | 00 | 37 32 31 21 21 21 21 21 |
| | | | | | | | | | P..... 721!!!! |
| TCP | 552 | 64062→9001 | [PSH, ACK] | Seq=513 | Ack=1 | Win=525568 | Len=512 | | Client's Public Key |
| 0020 | 50 | 18 | 08 | 05 | 9f | bb | 00 | 00 | 33 33 33 39 34 33 21 21 |
| | | | | | | | | | P..... 333943!! |
| TCP | 552 | 9001→64062 | [PSH, ACK] | Seq=1 | Ack=1025 | Win=525056 | Len=512 | | Client's N |
| 0020 | 50 | 18 | 08 | 03 | ae | e1 | 00 | 00 | 33 39 35 21 21 21 21 21 |
| | | | | | | | | | P..... 395!!!! |
| TCP | 552 | 9001→64062 | [PSH, ACK] | Seq=513 | Ack=1025 | Win=525056 | Len=512 | | Server's Public Key |
| 0020 | 50 | 18 | 08 | 03 | 99 | bb | 00 | 00 | 32 30 37 38 33 39 21 21 |
| | | | | | | | | | P..... 207839!! |
| TCP | 552 | 64062→9001 | [PSH, ACK] | Seq=1025 | Ack=1025 | Win=524544 | Len=512 | | Server's N |
| 0020 | 50 | 18 | 08 | 01 | aa | e9 | 00 | 00 | 33 33 35 21 21 21 21 21 |
| | | | | | | | | | P..... 335!!!! |
| TCP | 552 | 9001→64062 | [PSH, ACK] | Seq=1025 | Ack=1537 | Win=524544 | Len=512 | | Client's NONCE |
| 0020 | 50 | 18 | 08 | 01 | 6f | af | 00 | 00 | 30 30 31 30 31 30 31 30 |
| 0030 | 31 | 31 | 31 | 21 | 21 | 21 | 21 | 21 | 111!!!! |
| TCP | 552 | 9001→64062 | [PSH, ACK] | Seq=1537 | Ack=1537 | Win=524544 | Len=512 | | |
| 0020 | 50 | 18 | 08 | 01 | 6d | af | 00 | 00 | 30 30 31 30 31 30 31 30 |
| 0030 | 31 | 31 | 31 | 21 | 21 | 21 | 21 | 21 | 111!!!! |
| TCP | 552 | 9001→64062 | [PSH, ACK] | Seq=2049 | Ack=1537 | Win=524544 | Len=512 | | |
| 0020 | 50 | 18 | 08 | 01 | 70 | b0 | 00 | 00 | 30 30 30 30 30 30 30 30 |
| 0030 | 30 | 30 | 30 | 21 | 21 | 21 | 21 | 21 | 000!!!! |
| TCP | 552 | 9001→64062 | [PSH, ACK] | Seq=2561 | Ack=1537 | Win=524544 | Len=512 | | |
| 0020 | 50 | 18 | 08 | 01 | f4 | 00 | 00 | 00 | 39 31 36 39 30 40 39 31 |
| 0030 | 36 | 39 | 30 | 40 | 38 | 31 | 30 | 30 | 690@8100 1@183974 |
| 0040 | 40 | 39 | 36 | 39 | 38 | 31 | 40 | 39 | @96981@9 1690@969 |
| 0050 | 38 | 31 | 40 | 31 | 38 | 33 | 39 | 37 | 81@18397 4@96981@ |
| 0060 | 31 | 34 | 33 | 35 | 37 | 40 | 31 | 38 | 14357@18 315@!!!! |
| TCP | 552 | 9001→64062 | [PSH, ACK] | Seq=3073 | Ack=1537 | Win=524544 | Len=512 | | Server's Message |
| 0020 | 50 | 18 | 08 | 01 | 8f | bd | 00 | 00 | 31 35 31 38 38 34 21 21 |
| | | | | | | | | | P..... 151884!! |
| TCP | 552 | 64062→9001 | [PSH, ACK] | Seq=1537 | Ack=3585 | Win=525056 | Len=512 | | Server's NONCE |
| 0020 | 50 | 18 | 08 | 03 | 6a | 9f | 00 | 00 | 30 30 30 30 30 30 30 30 |
| 0030 | 30 | 30 | 30 | 21 | 21 | 21 | 21 | 21 | 0000!!!! |
| TCP | 552 | 64062→9001 | [PSH, ACK] | Seq=2049 | Ack=3585 | Win=525056 | Len=512 | | |
| 0020 | 50 | 18 | 08 | 03 | 68 | 9f | 00 | 00 | 30 30 30 30 30 30 30 30 |
| 0030 | 30 | 30 | 30 | 21 | 21 | 21 | 21 | 21 | 0000!!!! |
| TCP | 552 | 64062→9001 | [PSH, ACK] | Seq=2561 | Ack=3585 | Win=525056 | Len=512 | | |
| 0020 | 50 | 18 | 08 | 03 | 66 | 9f | 00 | 00 | 30 30 30 30 30 30 30 30 |
| 0030 | 30 | 30 | 30 | 21 | 21 | 21 | 21 | 21 | 0000!!!! |
| TCP | 552 | 64062→9001 | [PSH, ACK] | Seq=3073 | Ack=3585 | Win=525056 | Len=512 | | |
| 0020 | 50 | 18 | 08 | 03 | 7a | d0 | 00 | 00 | 32 39 34 31 34 38 40 33 |
| 0030 | 31 | 36 | 33 | 40 | 32 | 39 | 34 | 31 | P...z... 294148@3 |
| 0040 | 31 | 40 | 33 | 31 | 38 | 38 | 38 | 31 | 163@2941 48@31888 |
| 0050 | 39 | 31 | 36 | 34 | 36 | 40 | 32 | 34 | 1@318881 @113432@ |
| 0060 | 36 | 38 | 39 | 40 | 39 | 31 | 36 | 34 | 91646@24 7451@157 |
| 0070 | 33 | 31 | 36 | 33 | 40 | 21 | 21 | 21 | 689@9164 6@73917@ |
| | | | | | | | | | 3163@!!! |

그림 5.4 WireShark를 통한 PFS 프로토콜

그림 5.4는 WireShark를 통해 앞서 세션키를 생성하는 과정의 패킷들을 캡처한 모습이다. 우측의 설명을 보면 각각 어떤 내용의 패킷이 전달되는지 알 수 있다. 하지만 그림 5.4에서 붉은 박스로 표시된 부분의 패킷은 아직 설명하지 않았지만 간단하게 말하면 RSA 암호화의 Padding 정보이다. 이에 대한 내용은 바로 이어서 설명한다.

위의 시연에서는 RSA 공개키 교환과 PFS 프로토콜을 한 번에 보여주었지만 실제로는 따로 실행한다. 그 이유는 RSA 개인키를 마스터키로 하여 PFS 프로토콜을 사용해 세션키를 만드는데 세션키는 주기적으로 혹은 명시적으로 갱신 되어야하기 때문이다.

5.1. 보완된 RSA 알고리즘

직접 구현한 기존의 RSA 알고리즘은 평문의 각 문자에 대한 아스키코드 값으로 암호화 및 복호화를 수행하였다. 이는 고작 0~127에 해당하는 숫자를 다루는 것이므로 아무 무리 없이 동작하였다. 하지만 PFS 프로토콜을 구현하는 과정에서 암호화 후 서명된 데이터를 다시 서명을 풀고 복호화한 결과가 경우에 따라서 맞을 때도 있고 틀릴 때도 있는 문제가 발생하였다. 이를 해결하기 위해 BigInteger 라이브러리를 적용시켜 수의 오버플로우를 방지하기도 하고 여러 가지 가능성을 추측해보며 해결하려 했지만 번번이 실패하였다. 그러다 다시 RSA 알고리즘의 정의를 찾아보는 중, 암호화 될 숫자의 크기가 RSA 알고리즘을 위해 선택된 두 소수의 곱인 N보다 커서는 안 된다는 사실을 발견했다. 작은 수를 가지고 암호화를 한 번 수행하고 복호화를 수행하는 것은 무리가 없었지만 암호화가 수행 되어 커진 숫자를 다시 서명을 하려고 하니 문제가 생긴 것이었다.

이 문제를 해결하기 위해 데이터를 N이 넘지 않도록 Padding 시켜줄 필요가 있었다. 필자는 간단하게 N보다 수가 커진 데이터에 대하여 N보다 작아질 때까지 N을 빼주는 알고리즘을 구현하였다. 또한 올바른 복호화를 위해 N을 뺀 문자의 위치와 횟수 등의 정보를 저장하기 위한 배열을 사용했다.



그림 5.1.1 RSA Padding 정보

최초로 전송되는 배열은 발생한 에러의 종류를 확인하기 위한 정보이다. 예를 들어 배열 값이 1일 경우 서명할 때 에러가 발생할 것이라서 N을 차감했다는 뜻이고, 2일 경우 서명한 후, 즉 서명을 풀 때 에러가 발생할 것이라서 N을 차감했다는 뜻이다. 그리고 3일 경우 서명을 할 때와 서명을 풀 때 모두 에러가 발생할 것이라서 N을 차감했다는 뜻이다.

두 번째로 전송되는 배열은 서명할 때 N 을 차감한 횟수를 저장한다. 즉, N 을 몇 번 차감했는지 확인하여 수신측에서 서명을 풀 후 저장된 값만큼 N 을 반복해서 더한다.

마지막으로 전송되는 배열은 서명한 후, 서명을 풀 때 N 을 차감한 횟수를 저장하며 역시 수신측에서 서명을 풀 후 저장된 값만큼 N 을 반복해서 더한다.

송신측에서는 그림 5.1.1의 배열 정보를 전송하여 수신측에서 이 배열을 기반으로 N 을 다시 적당히 더하여 오류 없이 복호화가 가능했다. 다시 생각해보니 서명 후 에러는 송신측이 아닌 수신측에서 확인하여 처리가 가능하기 때문에 세 번째 배열은 전송할 필요가 없다.

6. 트루디의 공격 시연

6.1. 재전송 공격 테스트

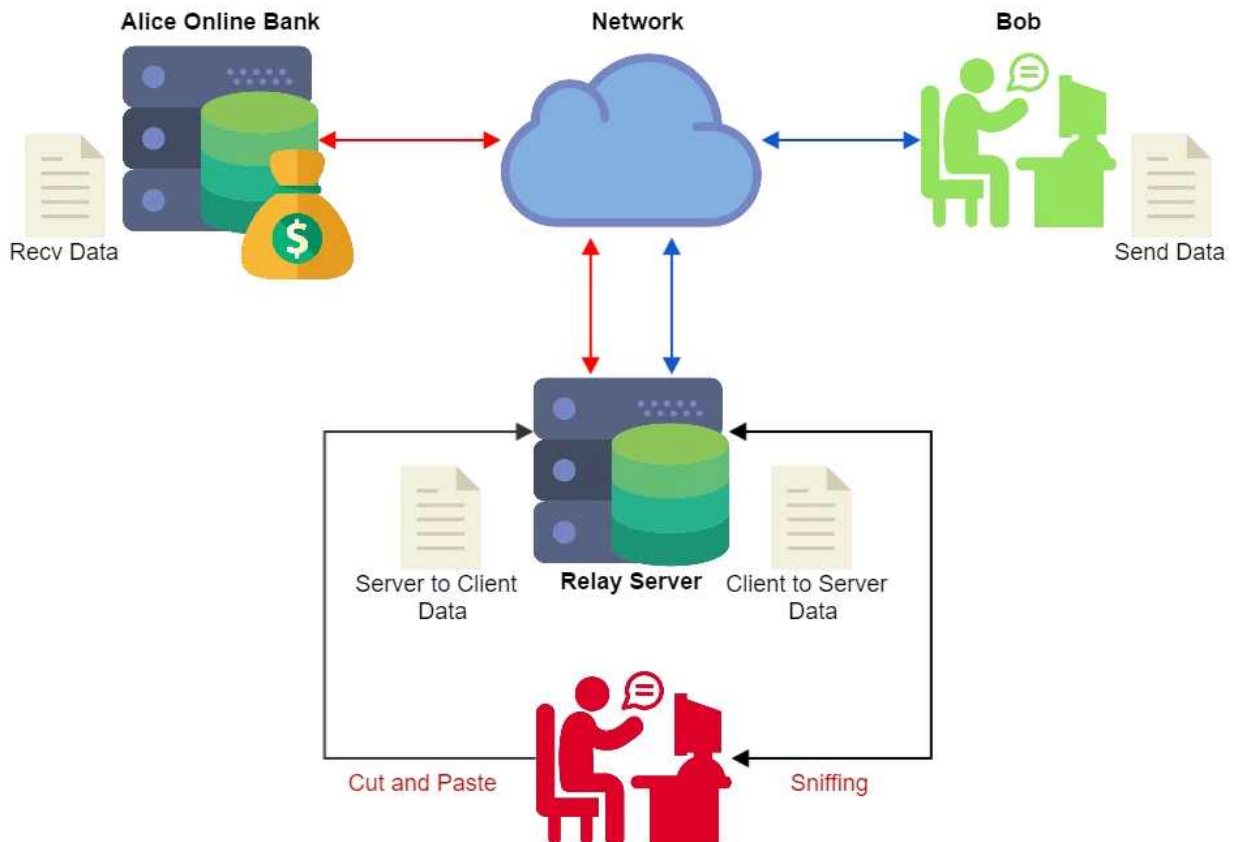


그림 6.1.1 Replay Attack Diagram

이번 장에서는 트루디가 릴레이 서버에 접속하여 클라이언트에서 서버로 전송되는 데이터를 변조해보는 테스트를 시연해본다. 테스트는 A5/1 스트림 암호화를 통해 암호화 전송 및 수신에 적용되고 ECC 디피-헬먼 알고리즘을 사용하여 키를 교환하는 시스템에서 시연된다. 그리고 시연 시스템은 클라이언트 측만 인증을 하는 단방향 인증을 지원하고 단지 패스워드를 암호화만 할 뿐 다른 프로토콜은 적용되지 않았다. 따라서 트루디에 의해 재전송 공격 및 중간자 공격이 가능하다. 우선 재전송 공격을 보이고 이 공격을 방지할 수 있도록 시스템을 보완한다. 그리고 보완된 시스템에 대하여 중간자 공격을 시도하는 것을 보인다. 마지막으로 서버 측 사용자 인증 정보 파일은 해시 되지 않은 상태로 진행한다. 그 이유는 어떤 정보가 현재 들어있는지 보면서 시연하는 것이 이번 테스트 환경으로 적합하다고 판단했기 때문이다.

트루디의 재전송 공격을 조금 더 간단하게 살펴보면 그림 6.1.2와 같다.

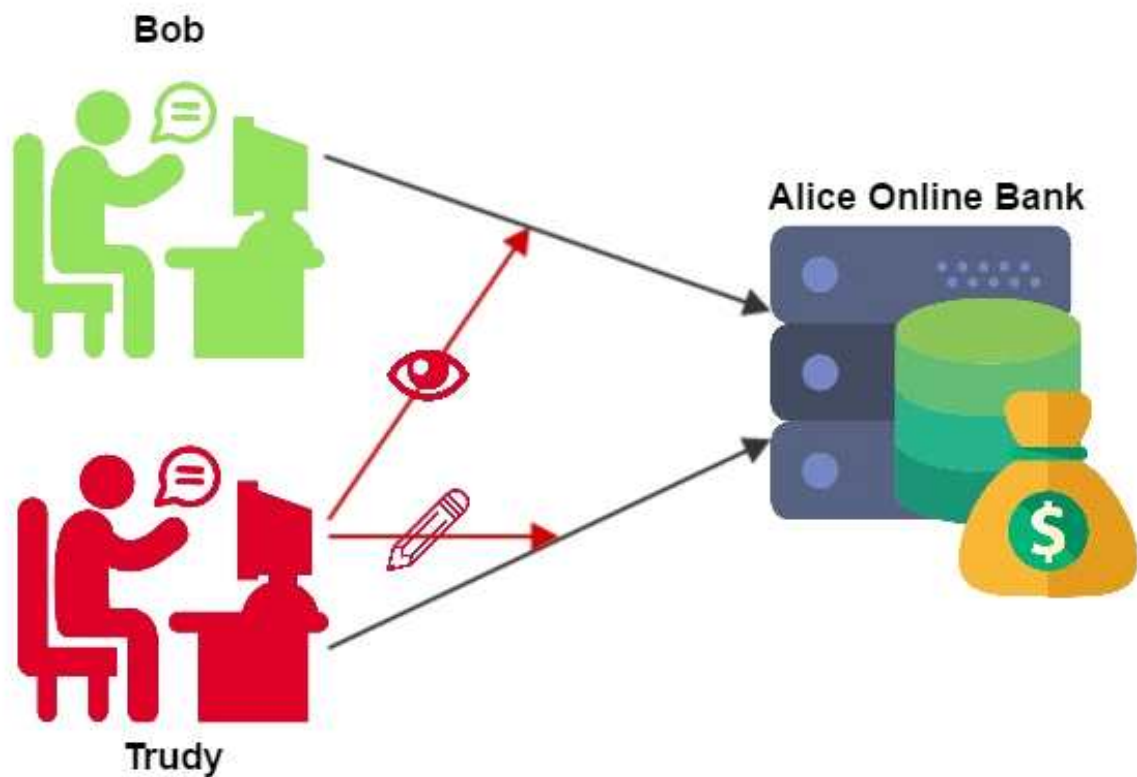


그림 6.1.2 Replay Attack Simple Diagram

Bob이 로그인 할 때 전송되는 정보를 트루디가 훔쳐보아 그 정보를 복사한 다음 자신이 로그인할 때 복사해 둔 정보를 Cut and Paste함으로써 자신이 Bob의 계정으로 로그인하는 공격법이다.

우선 아래의 그림 6.1.3은 Bob이 로그인할 때 트루디는 릴레이 서버에 접속하여 전송되는 정보를 훑쳐보는 화면이다.

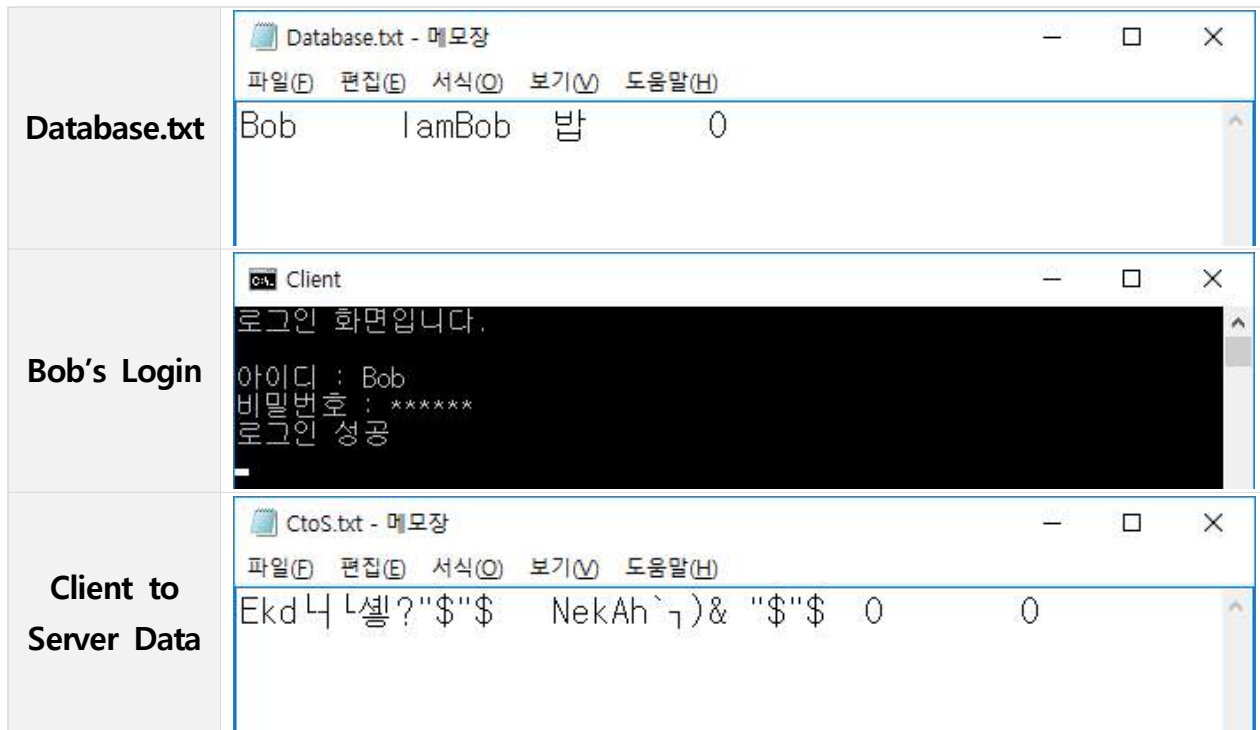


그림 6.1.3 Bob의 로그인 화면(Trudy가 훑쳐봄)

| Bob's Login | | | | | | | | | | | | | | | | | | |
|------------------|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----------|----------|
| Client to Server | | | | | | | | | | | | | | | | | | |
| TCP | 552 49847→9000 [PSH, ACK] Seq=1 Ack=1 Win=252 Len=512 | | | | | | | | | | | | | | | | | |
| 0020 | 50 | 18 | 00 | fc | e5 | ba | 00 | 00 | 45 | 6b | 64 | 03 | 17 | 03 | 99 | 8f | P..... | Ekd..... |
| 0030 | f0 | 7e | 03 | 05 | 03 | 05 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | .~....!! | !!!!!!! |
| TCP | 552 49847→9000 [PSH, ACK] Seq=513 Ack=1 Win=252 Len=512 | | | | | | | | | | | | | | | | | |
| 0020 | 50 | 18 | 00 | fc | 89 | 2b | 00 | 00 | 4e | 65 | 6b | 41 | 68 | 60 | 02 | 08 | P....+.. | NekAh`.. |
| 0030 | 07 | 01 | 03 | 05 | 7d | 04 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |}.!! | !!!!!!! |
| Server to Client | | | | | | | | | | | | | | | | | | |
| TCP | 552 49848→9001 [PSH, ACK] Seq=1 Ack=1 Win=252 Len=512 | | | | | | | | | | | | | | | | | |
| 0020 | 50 | 18 | 00 | fc | 7d | 0d | 00 | 00 | 45 | 6b | 64 | 03 | 17 | 03 | 99 | 8f | P...}... | Ekd..... |
| 0030 | f0 | 7e | 03 | 05 | 03 | 05 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | .~....!! | !!!!!!! |
| TCP | 552 49848→9001 [PSH, ACK] Seq=513 Ack=1 Win=252 Len=512 | | | | | | | | | | | | | | | | | |
| 0020 | 50 | 18 | 00 | fc | 20 | 7e | 00 | 00 | 4e | 65 | 6b | 41 | 68 | 60 | 02 | 08 | P... ~.. | NekAh`.. |
| 0030 | 07 | 01 | 03 | 05 | 7d | 04 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 |}.!! | !!!!!!! |

그림 6.1.4 WireShark를 통한 Bob의 로그인 패킷 분석

그림 6.1.4를 보면 클라이언트에서 릴레이 서버, 릴레이 서버에서 서버로 암호화된 ID와 PW가 전송되는 것을 볼 수 있다. 트루디는 이러한 정보를 릴레이 서버에 침입하여 가로챈다. 그 후 자신이 로그인할 때 전송되는 정보에 Bob의 암호화된 아이디와 패스워드를 붙여넣기 하여 전송한다. 실제 구현은 Sleep() 함수를 사용해 시간 딜레이를 준 다음 파일의 내용을 바꾸었다.

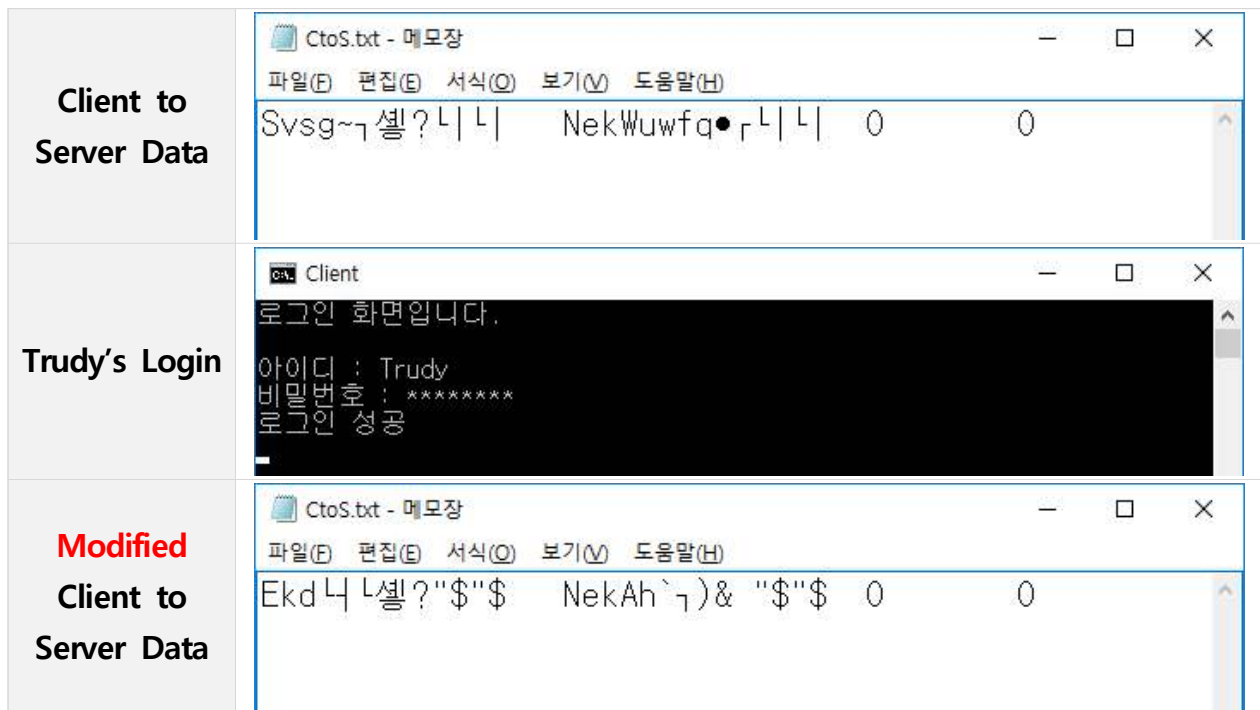


그림 6.1.5 트루디의 Cut and Paste Attack 화면

| Trudy's Login | |
|------------------|---|
| Client to Server | |
| TCP | 552 50049→9000 [PSH, ACK] Seq=1 Ack=1 Win=252 Len=512 |
| 0020 | 50 18 00 fc 90 b0 00 00 53 76 73 67 7e 02 99 8f P..... Svsg~... |
| 0030 | f0 7e 03 05 03 05 21 21 21 21 21 21 21 21 21 21 .~....!! !!!!!!! |
| TCP | 552 50049→9000 [PSH, ACK] Seq=513 Ack=1 Win=252 Len=512 |
| 0020 | 50 18 00 fc c2 f9 00 00 4e 65 6b 57 75 77 66 71 P..... NekWuwfq |
| 0030 | 07 01 03 05 01 04 21 21 21 21 21 21 21 21 21 21!! !!!!!!! |
| Server to Client | |
| TCP | 552 50050→9001 [PSH, ACK] Seq=1 Ack=1 Win=252 Len=512 |
| 0020 | 50 18 00 fc 7d 0d 00 00 45 6b 64 03 17 03 99 8f P...}... Ekd..... |
| 0030 | f0 7e 03 05 03 05 21 21 21 21 21 21 21 21 21 21 .~....!! !!!!!!! |
| TCP | 552 50050→9001 [PSH, ACK] Seq=513 Ack=1 Win=252 Len=512 |
| 0020 | 50 18 00 fc 20 7e 00 00 4e 65 6b 41 68 60 02 08 P... ~.. NekAh`.. |
| 0030 | 07 01 03 05 7d 04 21 21 21 21 21 21 21 21 21 21}.!! !!!!!!! |

그림 6.1.6 WireShark를 통한 트루디의 Cut and Paste Attack 패킷 분석

그림 6.1.6을 보면 클라이언트에서 릴레이 서버로의 메시지와 릴레이 서버에서 서버로의 메시지가 다른 것을 볼 수 있다. 이는 트루디가 릴레이 서버에 침입하여 데이터를 변조했기 때문이다. 따라서 트루디는 로그인 화면에서 서버에 등록되지 않은 정보를 입력했지만, 릴레이 서버에 접속하여 전송되는 데이터를 변조함으로써 Bob의 계정으로 로그인할 수 있게 되었다. 스트림 암호이기 때문에 암호문과 평문의 길이가 같아야 하는데 위의 예에서는 다르다. 그 이유는 암호화 및 복호화를 평문의 길이만큼 해주지 않았기 때문이다. 아래부터는 수정 후 테스트한 결과이므로 평문과 암호문의 길이가 같다.

6.2. 재전송 공격 방지 테스트

다음으로 트루디의 Replay Attack을 방지하는 테스트를 해본다. 구현은 서버에서 클라이언트에게 제공한 NONCE를 사용해 클라이언트에서 $E((PW, NONCE), KEY)$ 방식으로 암호화하여 전송한다. 서버 측에서는 암호문을 받아 해독 후 자신이 전송했던 NONCE와 비교하여 일치한다면 그 이후에 ID 및 PW 인증 과정을 시작한다. 이 NONCE는 rand() 함수를 통해 매번 바뀌기 때문에 트루디는 단순히 암호문 자체를 저장한 후 그대로 사용하는 Replay Attack을 수행하지 못할 것이다. 물론 NONCE를 그렇게 간단하게 이어 붙인다면 보안상 어느 정도 유추가 가능하기 때문에 보안상 위험하지만 여기서는 트루디가 암호문을 단순히 Cut and Paste 함으로써 Bob의 계정으로 로그인 할 수 없다는 것을 보이는 데 초점을 맞추었다.

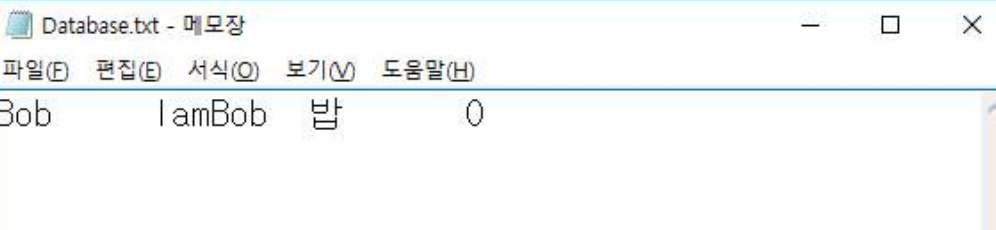
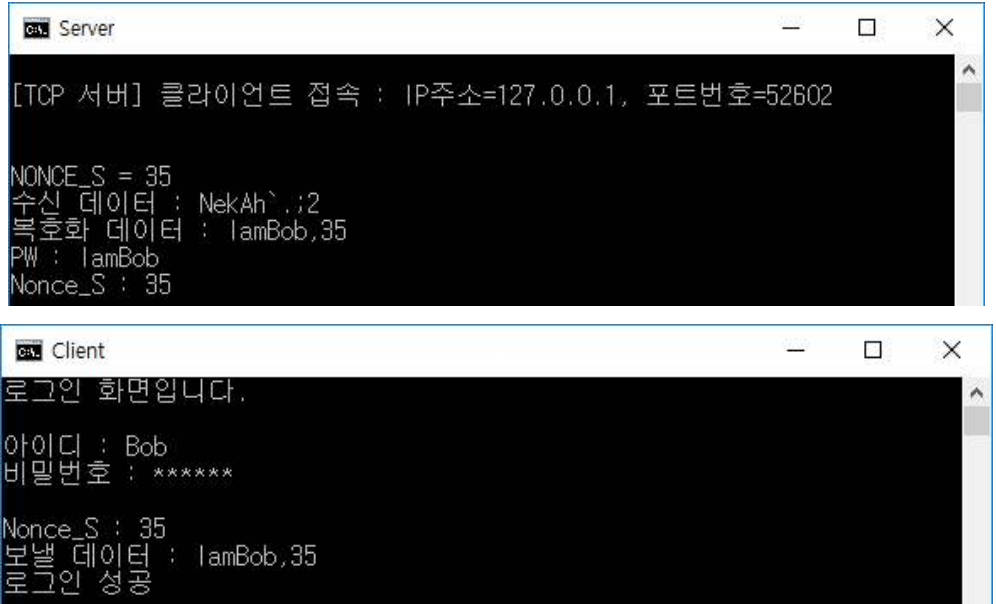
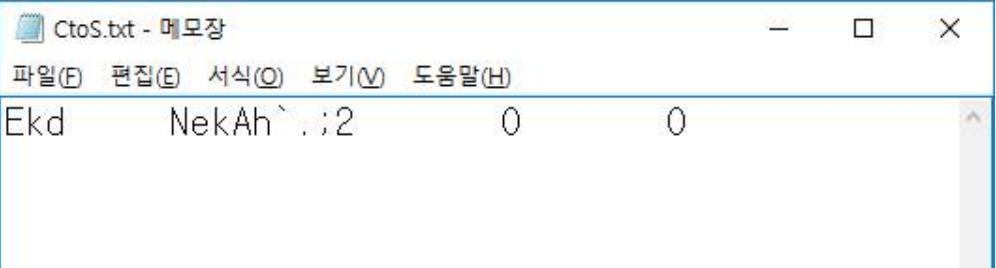
| | |
|-----------------------|--|
| Database.txt |  |
| Bob's Login |  |
| Client to Server Data |  |

그림 6.2.1 NONCE 기법 적용 후 Bob의 로그인 화면(Trudy가 훔쳐봄)

그림 6.2.1은 NONCE 기법이 적용된 후 Bob이 로그인하는 화면이다. 트루디는 릴레이 서버에 칩입한 후 Bob의 암호화된 패스워드를 저장했다고 가정하자. 우선 아래의 그림 6.2.2은 Bob이 로그인할 때마다 암호문이 NONCE에 의해 다르다는 것을 보여주는 화면이다.

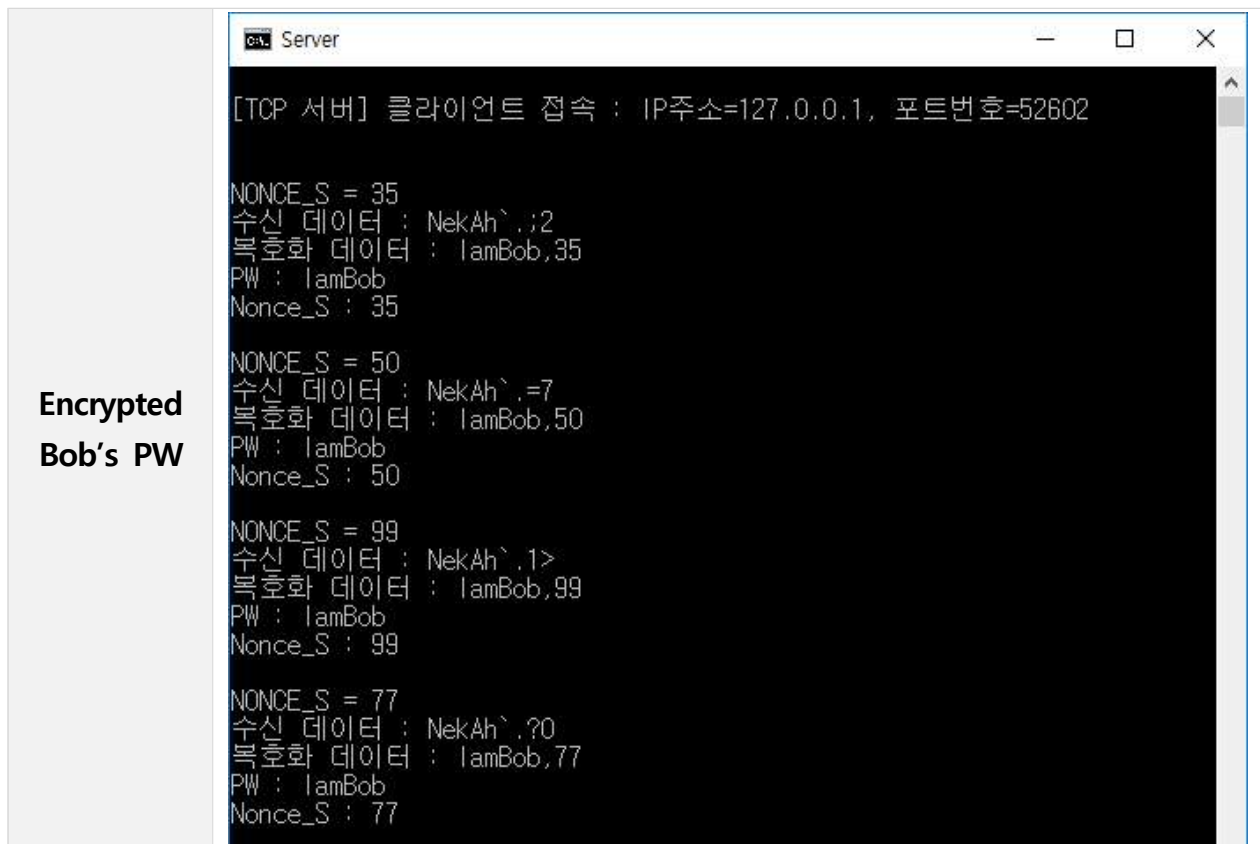


그림 6.2.2 매번 다른 암호화된 Bob의 패스워드

| Bob's Login | |
|------------------|---|
| Client to Server | |
| TCP | 552 52678→9000 [PSH, ACK] Seq=1537 Ack=1537 Win=525568 Len=512 |
| 0020 | 50 18 08 05 15 21 00 00 45 6b 64 21 21 21 21 21 P....!.. Ekd!!!! |
| TCP | 552 52678→9000 [PSH, ACK] Seq=2049 Ack=1537 Win=525568 Len=512 |
| 0020 | 50 18 08 05 9d ad 00 00 4e 65 6b 41 68 60 2e 3b P..... NekAh`.; |
| 0030 | 32 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 2!!!!!!! !!!!!!!! |
| Server to Client | |
| TCP | 552 52679→9001 [PSH, ACK] Seq=1537 Ack=1537 Win=525568 Len=512 |
| 0020 | 50 18 08 05 a1 0c 00 00 45 6b 64 21 21 21 21 21 P..... Ekd!!!! |
| TCP | 552 52679→9001 [PSH, ACK] Seq=2049 Ack=1537 Win=525568 Len=512 |
| 0020 | 50 18 08 05 29 99 00 00 4e 65 6b 41 68 60 2e 3b P...)... NekAh`.; |
| 0030 | 32 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 2!!!!!!! !!!!!!!! |

그림 6.2.3 WireShark를 통한 Bob의 로그인 패킷 분석

트루디는 그림 6.2.3의 패킷을 스니핑하여 복사한 후 자신이 로그인할 때 패킷을 변조하려고 한다. 아래의 그림 6.2.4는 암호문을 그대로 복사하여 자신이 로그인할 때 붙여넣기 하였지만 로그인에는 실패하는 모습이다.

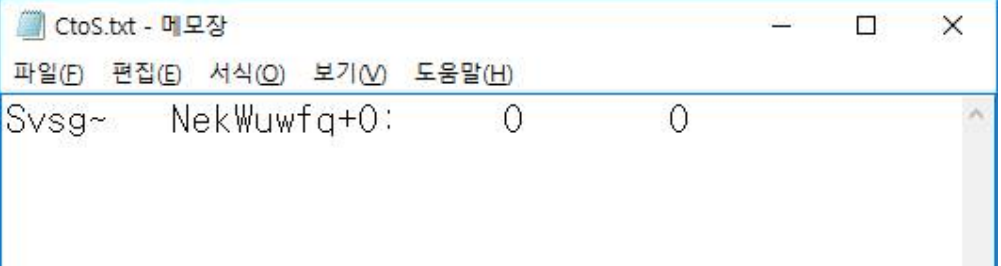
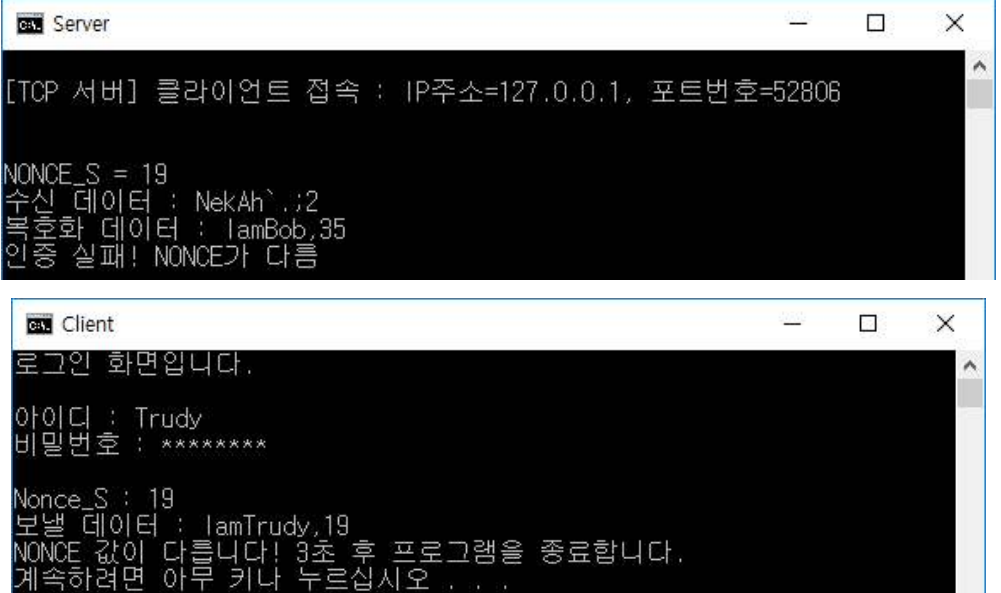
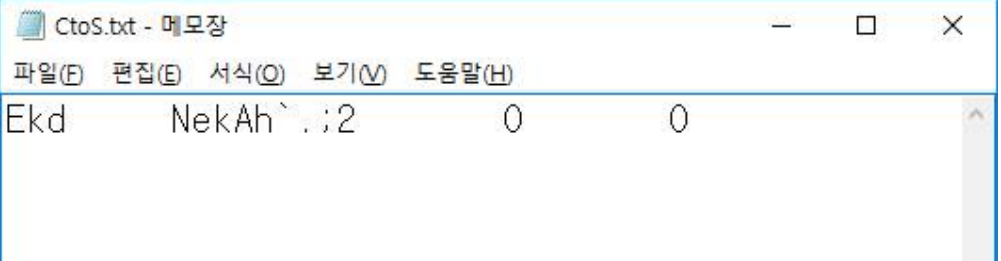
| | |
|--------------------------------|--|
| Client to Server Data |  |
| Trudy's Login |  |
| Modified Client to Server Data |  |

그림 6.2.4 트루디의 Cut and Paste Attack 실패 화면

| Trudy's Login | | | | | | | | | | | | | | |
|------------------|-----|-------|----|------|------------|----------|---------|----------|---------|----|----|----|----|-----------------------------|
| Client to Server | | | | | | | | | | | | | | |
| TCP | 552 | 52805 | → | 9000 | [PSH, ACK] | Seq=513 | Ack=513 | Win=2053 | Len=512 | | | | | |
| 0020 | 50 | 18 | 08 | 05 | 30 | 31 | 00 | 00 | 53 | 76 | 73 | 67 | 7e | 21 21 21 P...01.. Svsg~!!! |
| TCP | 552 | 52805 | → | 9000 | [PSH, ACK] | Seq=1025 | Ack=513 | Win=2053 | Len=512 | | | | | |
| 0020 | 50 | 18 | 08 | 05 | db | 9c | 00 | 00 | 4e | 65 | 6b | 57 | 75 | 77 66 71 P..... NekWuwfq |
| 0030 | 2b | 30 | 3a | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 +0:!!!!!! !!!!!!!! |
| Server to Client | | | | | | | | | | | | | | |
| TCP | 552 | 52806 | → | 9001 | [PSH, ACK] | Seq=513 | Ack=513 | Win=2049 | Len=512 | | | | | |
| 0020 | 50 | 18 | 08 | 01 | 59 | c7 | 00 | 00 | 45 | 6b | 64 | 21 | 21 | 21 21 21 P...Y... Ekd!!!!!! |
| TCP | 552 | 52806 | → | 9001 | [PSH, ACK] | Seq=1025 | Ack=513 | Win=2049 | Len=512 | | | | | |
| 0020 | 50 | 18 | 08 | 01 | e2 | 53 | 00 | 00 | 4e | 65 | 6b | 41 | 68 | 60 2e 3b P....S.. NekAh`.; |
| 0030 | 32 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 | 21 2!!!!!!! !!!!!!!! |

그림 6.2.5 WireShark를 통한 트루디의 Cut and Paste Attack 패킷 분석

그림 6.2.4에서 서버는 NONCE로 19를 요구했지만 트루디는 이전에 Bob이 자신의 패스워드와 NONCE 값 35를 사용해 암호화했던 암호문을 그대로 사용했다. 따라서 서버는 자신이 요구한 19와 다르기 때문에 인증을 거절하는 것을 볼 수 있다.

6.3. 중간자 공격 테스트

하지만 여전히 중간자 공격은 가능하다. 아래의 그림 6.3.1는 중간자 공격이 동작하는 과정을 나타낸 다이어그램이다.

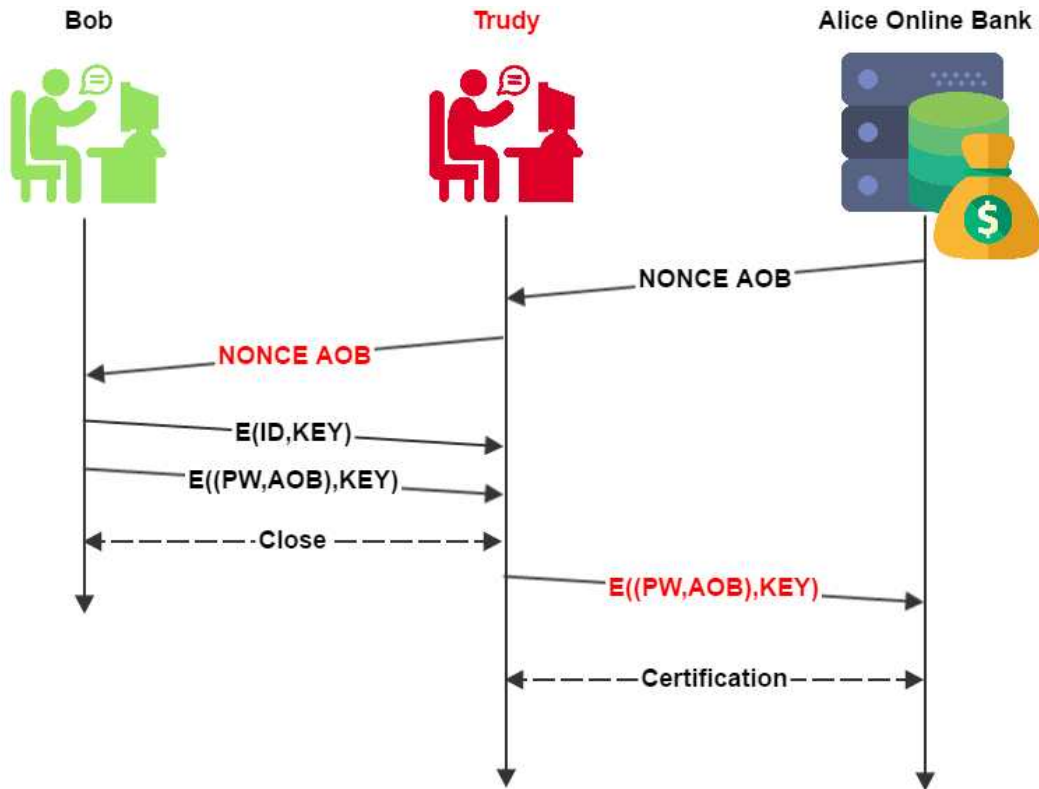


그림 6.3.1 중간자 공격 다이어그램

그림 6.3.1를 천천히 살펴보자. 우선 트루디는 AOB 서버에게 인증을 요청한다. 서버는 트루디의 재전송 공격을 막기 위해 NONCE를 전송한다. 하지만 트루디는 Bob의 계정과 NONCE가 섞인 암호문을 만들어낼 수 없다. 따라서 암호문을 만들어내는 작업을 Bob에게 떠넘긴다. 트루디는 Bob의 암호 혹은 비밀키를 알 수 없지만 어쨌든 Bob의 암호와 서버가 요청한 NONCE가 섞인 암호문을 얻어낼 수 있다. 그 암호문을 다시 서버로 전송함으로써 트루디는 Bob의 계정으로 로그인할 수 있게 된다.

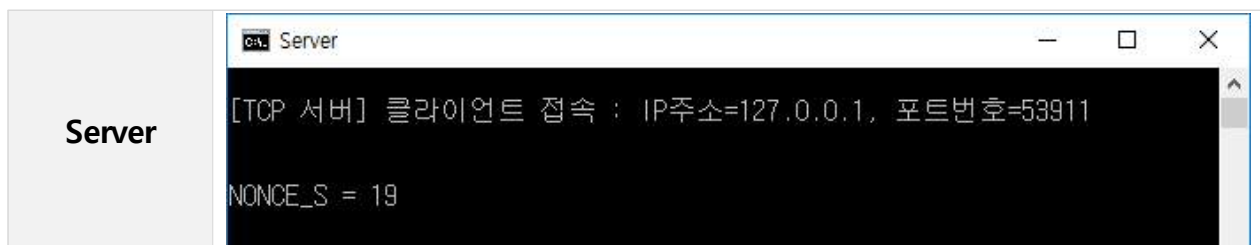


그림 6.3.2 서버가 NONCE 값으로 19를 요구하는 화면

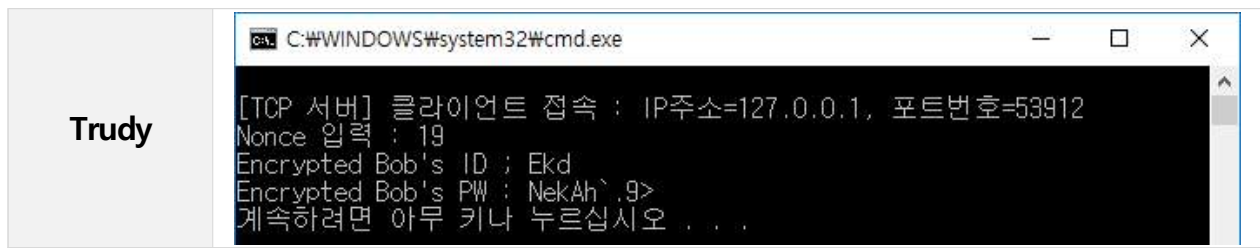


그림 6.3.3 Trudy가 Bob에게 NONCE 값으로 19를 주고 암호문을 받는 화면

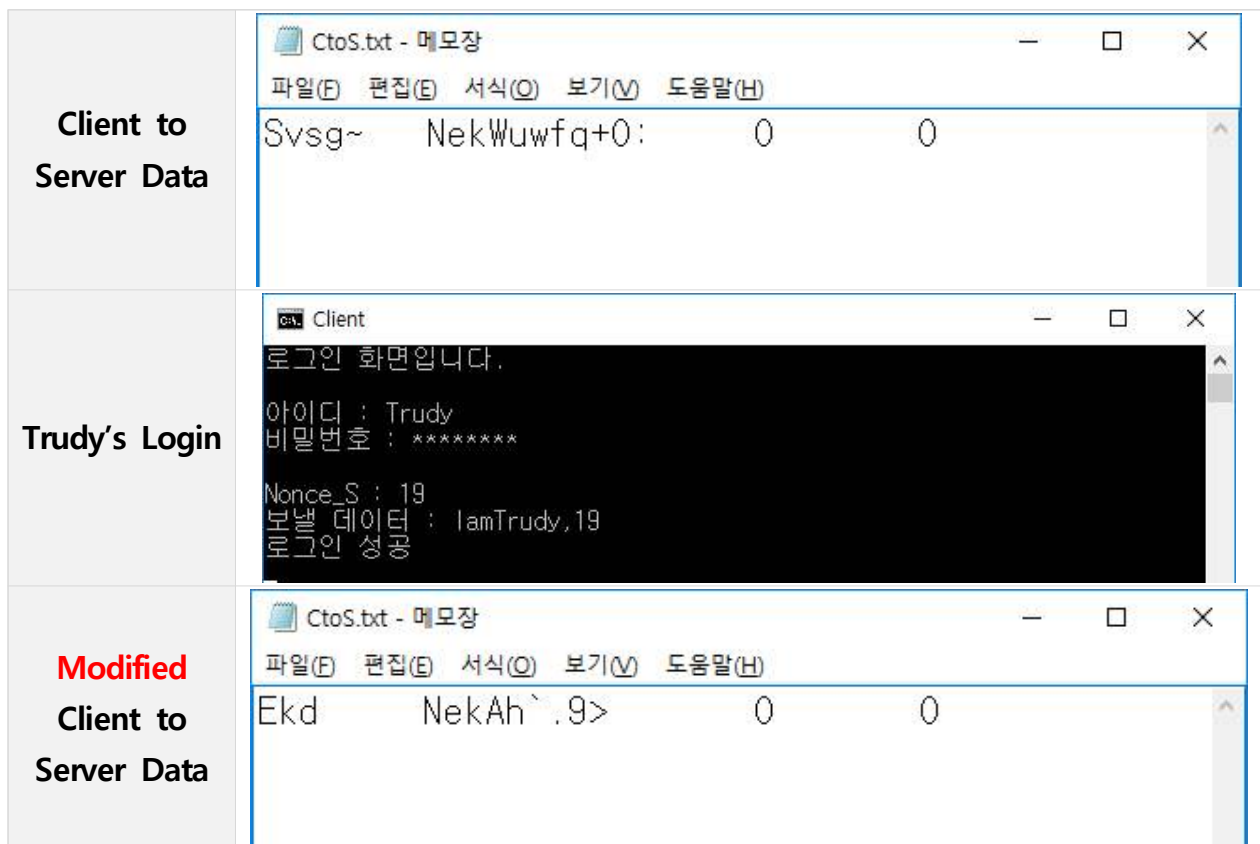


그림 6.3.4 트루디의 Cut and Paste Attack 화면

그림 6.3.2~4를 보면 NONCE를 사용해 기본적인 재전송 공격을 방지할 수 있었지만 중간자 공격은 여전히 방지할 수 없다는 것을 볼 수 있다. 그렇다면 중간자 공격을 막기 위해서는 어떻게 해야 할까?

우선 교과서와는 예시가 조금 다르지만 위의 경우에서 중간자 공격을 막기 위해서는 상호 인증이 필요하다. 즉, 클라이언트가 서버가 올바른 서버인지 인증할 필요가 있다. 재전송 공격과 중간자 공격을 막기 위해서는 상호 인증과 암호화된 메시지 안에 자신임을 식별할 수 있는 내용과 NONCE가 함께 들어있어야 한다. 그리고 추가적인 보안성을 위해 보안 프로토콜은 세션키와 PFS를 만족해야한다. 즉 5장에서 보인 PFS 프로토콜을 사용하면 이번 6장에서 보인 공격법은 모두 방지할 수 있다.

7. 회고

이번 장에서는 최종 시스템으로 발전하기까지의 보안상 문제점에 대해 알아보고, 구현하는 동안 어려웠던 점 그리고 배운 점에 대해 살펴본다.

우선 최초 시스템의 문제점은 아무런 보안작업이 없었기 때문에 인증 문제, 무결성 문제, 그리고 무엇보다도 최소한의 비밀성 또한 보장되지 않았다는 점이다. 따라서 패킷 스니핑에 의해 간단하게 모든 비밀 정보가 노출되었다.

| 스니핑 공격 | | |
|--|-----------------|--|
| 40 9000→55467 [ACK] Seq=307957336 Ack=4 Win=525568 Len=0 | | |
| 50 55467→9000 [PSH, ACK] Seq=4 Ack=307957336 Win=525568 Len=10 | | |
| 50 18 08 05 96 b1 00 00 72 6b 66 61 6f 72 6c 30 | P..... rkfaorl0 | |
| 50 30 | 00 | |
| 40 9000→55467 [ACK] Seq=307957336 Ack=14 Win=525568 Len=0 | | |
| 49 55467→9000 [PSH, ACK] Seq=14 Ack=307957336 Win=525568 Len=9 | | |
| 50 18 08 05 94 d7 00 00 72 6b 66 61 6f 72 6c 31 | P..... rkfaorl1 | |
| 32 | 2 | |

그림 7.1 스니핑 공격

패스워드가 그대로 노출된다는 것은 하나의 계정이 노출되는 것보다 훨씬 더 위험한 일일 수도 있다. 보통 사용자들은 하나의 패스워드를 여러 사이트에서 사용하기 때문이다.

따라서 최우선적으로 비밀성을 보장하기 위해 데이터를 암호화해서 보내는 방법을 사용했다.

실패한 스니핑 공격 (RSA 암호화)

```

40 9000→55777 [ACK] Seq=117 Ack=235 Win=525312 Len=0
99 55777→9000 [PSH, ACK] Seq=235 Ack=117 Win=525312 Len=59
04 70 34 00 00 32 32 36 35 32 40 32 30 P...p4.. 22652@20
40 35 34 36 34 32 40 33 31 38 32 40 35 793@5464 2@3182@5
33 40 32 32 36 35 32 40 33 30 33 32 38 7613@226 52@30328
30 35 32 40 35 38 30 35 32 40 35 38 30 @58052@5 8052@580
52@

40 9000→55777 [ACK] Seq=117 Ack=294 Win=525056 Len=0
93 55777→9000 [PSH, ACK] Seq=294 Ack=117 Win=525312 Len=53
04 20 85 00 00 32 32 36 35 32 40 32 30 P... .. 22652@20
40 35 34 36 34 32 40 33 31 38 32 40 35 793@5464 2@3182@5
33 40 32 32 36 35 32 40 33 30 33 32 38 7613@226 52@30328
32 37 39 40 31 36 37 32 39 40 @48279@1 6729@

```

그림 7.2 실패한 스니핑 공격 (RSA 암호화)

그 결과 단순히 스니핑을 하는 것으로는 패스워드를 확인할 수 없었다. 하지만 트루디가 실제 서버와 유사하게 동작하는 서버를 만들어서 클라이언트를 속일 수 있는 문제가 있었다. 이 공격법은 중간자 공격으로 응용될 수 있다. 이는 클라이언트가 서버를 인증하지 않기 때문에 생기는 문제점이다.

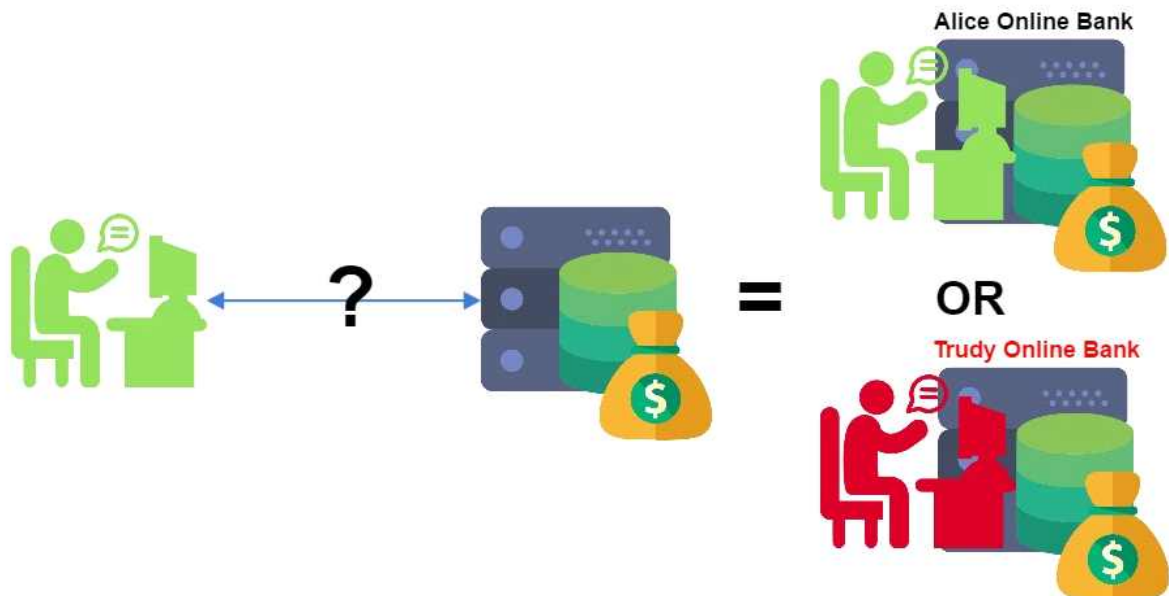


그림 7.3 서버 측 사용자 인증의 필요성

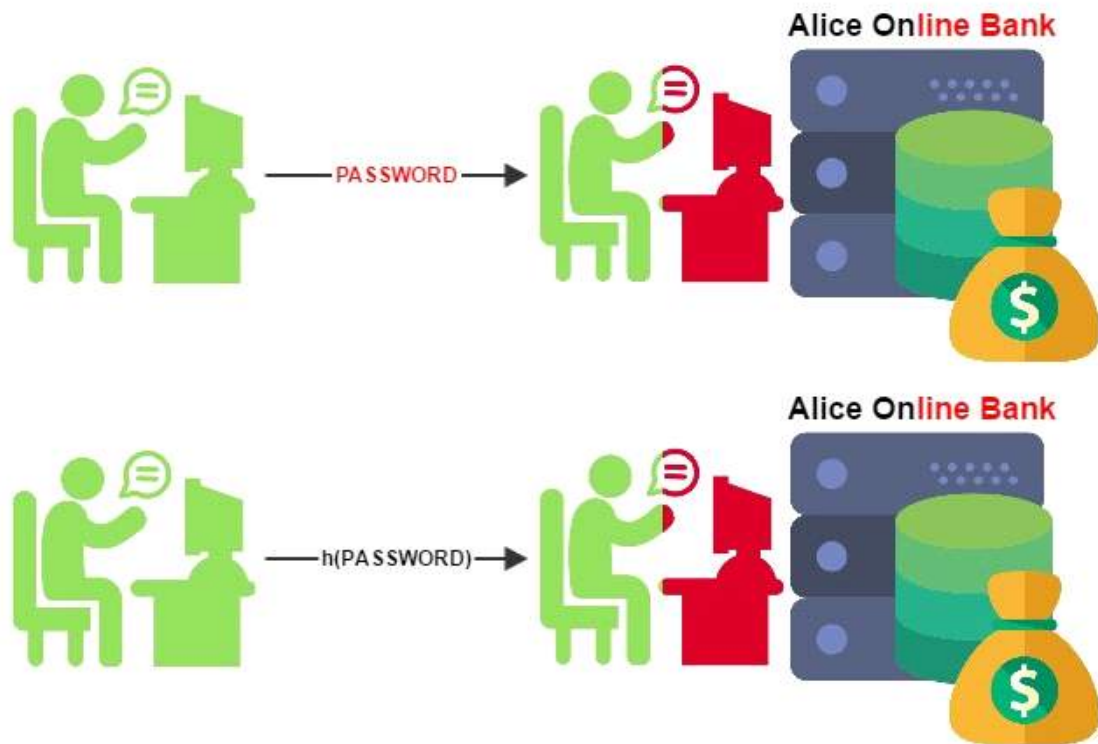


그림 7.5 신뢰할 수 없는 서버

하지만 트루디의 목적은 암호문을 푸는 것이 아니라 자신이 부당한 이익을 취하는 것이다. 따라서 트루디가 자신이 이익을 취하기 위해 암호문을 통째로 사용하는 문제가 생기게 된다. 6장 테스트에서 보았듯이 트루디가 중간에서 메시지를 변조하면 시스템의 규칙에 위반되는 상황을 여전히 쉽게 만들 수 있다. 6장 테스트에서는 전송 시 해시 함수가 적용되지는 않았지만, 이 재전송 공격은 해시를 해서 보내도 결과는 같다. 대비책은 어떤 것이 있을까? 그 중 하나로 NONCE 기법이 있다. 서버에서 제공하는 NONCE라는 변칙적인 값을 개인이 패스워드를 해시할 때 첨가하여 해시하는 방법이다. 이는 SALT 방법과 유사하다. 이렇게 하면 6장 테스트와 같이 지나가는 정보를 복사해서 보관해봤자 다시 사용할 수 없게 되므로 재전송 공격을 막을 수 있게 된다.

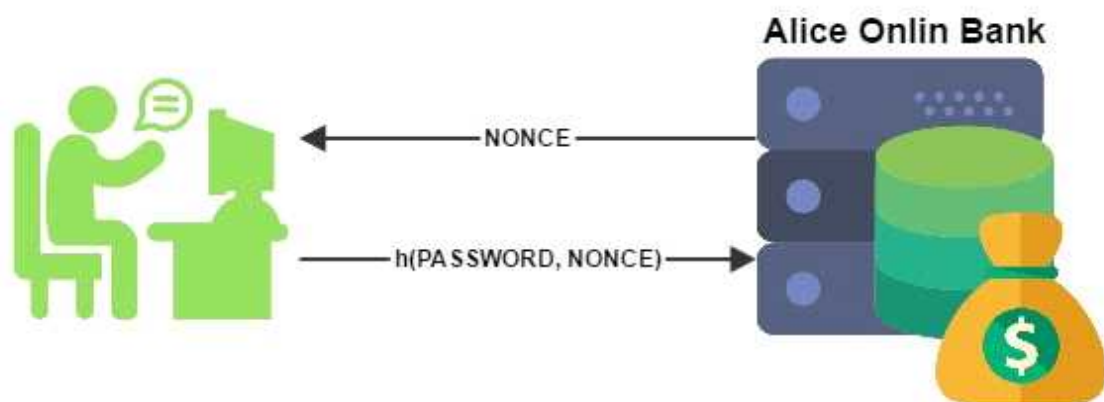


그림 7.6 NONCE 기법

하지만 이 방법은 서버 측이 개인의 패스워드를 알고 있는 상황에서 동작한다. 따라서 바로 이전에 말했던 문제점이 다시 제기되는 것이다. 그래서 이 문제와 트루디의 재전송 공격을 동시에 막는 것은 NONCE 값과 해시된 패스워드를 같이 해시함으로써 가능하다.

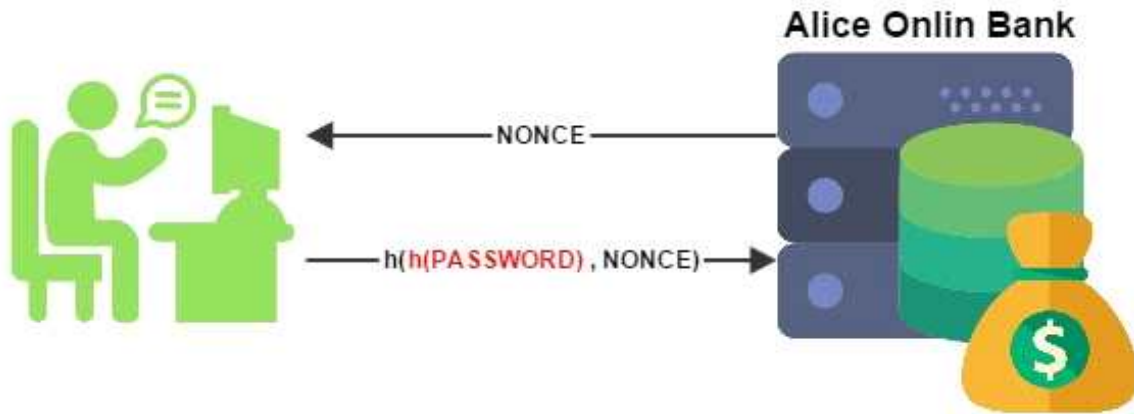


그림 7.7 Hashed NONCE 기법

또, 통신 내용을 암호화 한다고 해서 영원히 안전한 것은 아니다. 언젠가 비밀키가 노출될 수도 있다. 만약 트루디가 통신 내용을 모두 저장해두는 수고를 해왔다면 이 통신 내용들은 모두 해독되어질 것이다. 따라서 하나의 비밀키를 처음부터 끝까지 사용하는 것이 아닌 세션마다 다른 키를 사용하는 세션키 방식이 필요하다. 또한 이 세션키들 사이에는 연관성이 없어야 한다. 그 이유는 하나의 세션키가 노출되었을 때 키들 사이에 연관성이 있다면 다른 세션키로 암호화된 통신 내용 또한 쉽게 해독되어질 가능성이 있기 때문이다. 이 '세션키들 사이에는 연관성이 없어야 한다'라는 것을 우리는 Perfect Forward Secrecy, PFS라고 부른다. 따라서 이 PFS 또한 보장되어야 한다.

조금 앞서 설명한 상호 인증과 세션키, PFS를 모두 만족시키기 위해 사용된 보안 프로토콜이 바로 5장에서 설명한 PFS 프로토콜이다. RSA 알고리즘의 암호화 기능, 상호 인증 및 중간자 공격을 방지하는 서명 기능 그리고 ECC 디피-헬먼 알고리즘의 키 교환 기능, NONCE를 사용한 재전송 공격 방지 기능을 섞어 놓은 프로토콜을 사용함으로써 안전한 통신을 할 수 있게 되었다.

또한 이 PFS 프로토콜의 동작 과정을 나타낸 다이어그램인 그림 5.1을 보면 교과서의 [그림 10-3] 단순화된 SSL과 동작 과정이 상당히 유사한 것을 볼 수 있다. 아래의 그림 7.8은 교과서의 [그림 10-3]과 같은 그림이다. 그리고 데이터 암호화 및 복호화 적용 또한 응용 프로그램의 수정이 필요하다는 공통점이 있다. 따라서 최종 프로젝트에서 구현된 PFS 프로토콜은 감히 SSL의 동작 과정이 매우 흡사한 프로토콜이라고 할 수 있다.

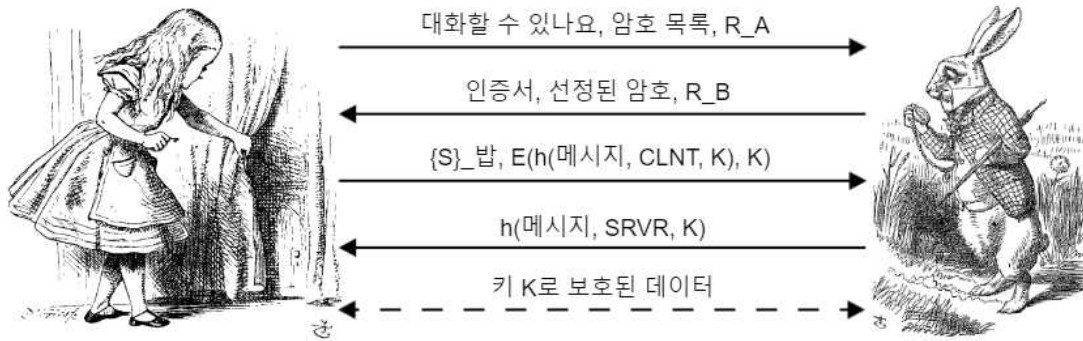


그림 7.8 단순화된 SSL

그리고 현 시스템은 소프트웨어 측면에서의 보안성을 전혀 고려하지 않았기 때문에 앞에서 설명하지 못한 취약점이 분명 존재한다. 예를 들면 버퍼 오버플로우 공격, 역어셈블 등이 가능하다. 또한 서버의 자원을 고갈시켜 서비스가 불가능하도록 하는 DoS 공격이 있을 수 있다. 이는 대비책을 세우기 굉장히 어려운데, L7 스위치가 DoS 공격을 어느 정도 막아낼 수 있다고 한다. 이 취약점들에 대한 조금 더 구체적인 대비책은 본 프로젝트에서는 생략했지만 추가로 조사하여 학습할 필요가 있을 것이다.

구현하면서 어려웠던 점은 일단 암호화 알고리즘을 적용할 때 비트 연산을 어떻게 프로그래밍에 적용시킬지가 개인적으로 조금 문제가 되었다. 하지만 조금 비효율적이지만 직관적으로 이해하기 쉽게 주로 정수형 배열을 사용해 구현하였다. 또 실제 다른 컴퓨터로 프로그램의 동작을 테스트할 때 TCP의 경계 때문에 제대로 동작되지 않는 문제도 발생하였지만, 고정 길이로 데이터를 송신하고 수신함으로써 해결하였다. 또 서버 측의 해시된 정보 파일을 읽을 때 텍스트로 읽어서 문제가 생겼지만 이는 바이너리로 읽음으로써 해결하였다. 또 비슷한 문제로 문자열을 암호화하니 간간히 암호문 사이에 원활한 동작을 방해하는 특수 문자가 들어가는 경우가 생기는 것을 운 좋게 파악하여 조건문으로 해결할 수 있었다. 그리고 RSA 알고리즘의 평문의 크기를 압축해주어야 한다는 사실을 알기까지 많은 고생을 했지만 Padding 알고리즘을 적용함으로써 해결할 수 있었다.

이번 과제를 진행하면서 여러 암호화 알고리즘들을 직접 구현해볼 수 있었으며, 직접 트루디가 되어 중간에서 정보를 변조하는 테스트도 해볼 수 있었다. 조금씩 진행되는 프로젝트 덕분에 전체적인 보안 개념을 학습하는 데 큰 도움이 되었다. 보안이라는 전체적인 그림과 여러 가지 공격법을 구사하는 트루디를 막기 위해 사용되는 여러 가지 기법들을 살펴보는 과정에서 많은 지식을 얻을 수 있었고 또한 보안을 공부하는 데 있어서 좋은 밑거름이 된 것 같아 뿌듯하다. 아직은 큰 틀을 배웠을 뿐이지만 추가적으로 학습해야 할 세부적인 내용들을 이해하는 데 큰 어려움이 없을 것이라 생각된다. 항상 남는 것이 많은 Term Project이다.

8. 부록

이번 장에서는 카이사르 암호에 대해 소개하고 암호문의 빈도수를 검사하여 Breaking 하는 프로그램을 보인다. 우선 카이사르 암호는 고전 암호 중 하나로서 1~25까지 25개의 키를 가진다. 암호화는 평문의 알파벳에 대하여 키의 값만큼 시프트 함으로써 수행되고, 복호화는 반대 방향으로 시프트 함으로써 수행된다. 이 암호화는 키의 범위가 매우 한정적이라는 점에서 취약하다. 따라서 전수검사의 비용이 크지 않지만 영 문장의 특성상 알파벳의 빈도수가 노출된다는 점을 흥미롭게 생각해서 프로그램으로 작성하여 보았다.

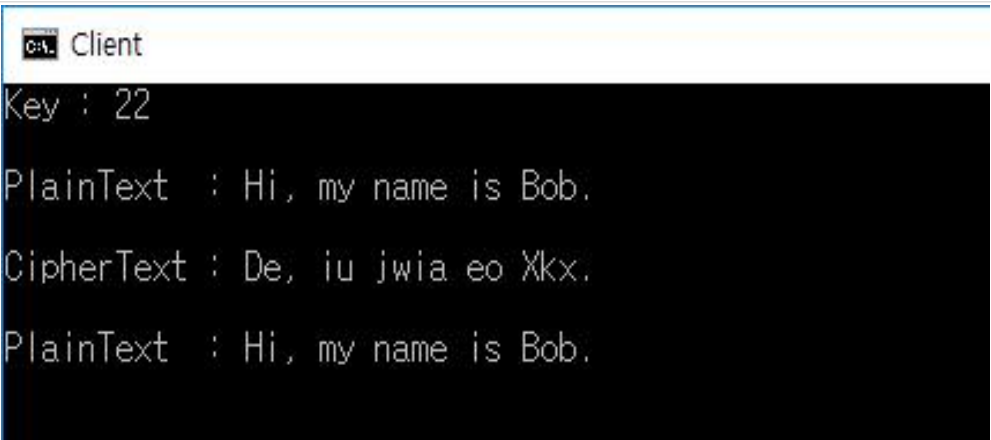
| | |
|-------------------------------|--|
| <p>카이사르 암호화 시연</p> |  |
| <p>카이사르 암호화 시연 코드</p> | <pre>int key = Key_Gender_Caesar(); strcpy(msg, "Hi, my name is Bob."); printf("Key : %d \n\n", key); printf("PlainText : %s \n\n", msg); Caesar(log, msg, key); printf("CipherText : %s \n\n", log); memset(msg, 0, strlen(msg)); Caesar(msg, log, -key); printf("PlainText : %s \n\n", msg);</pre> |

그림 8.1 카이사르 암호 시연 화면

Key : 22

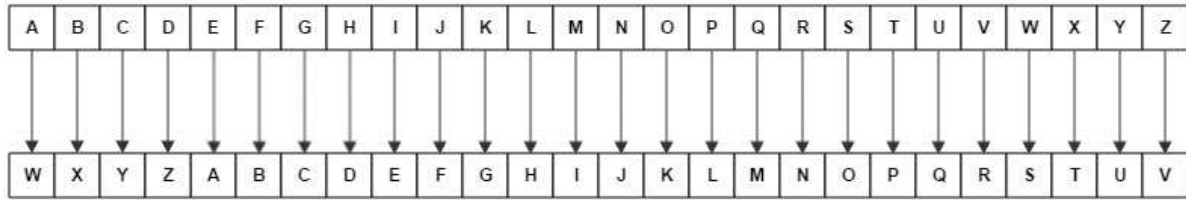


그림 8.2 key가 22일 때 카이사르 암호

이번에는 빈도수를 이용한 카이사르 암호 Breaking을 해보자. 우선 키는 난수로 지정해야 하지만 이번 테스트에서는 위의 다이어그램을 활용하기 위해 키 값을 22로 하여 에드거 앨런 포의 The Gold Bug 소설을 암호화한다.



그림 8.3 The Gold Bug 소설을 카이사르 암호화한 결과

이제 암호화가 된 텍스트 파일의 알파벳 빈도수를 프로그램을 사용해 세어볼 것이다. 그 전에 우리는 다음에 보이는 자료에서처럼 영어의 문장에서 알파벳 e가 가장 빈도수가 높은 것을 알고 있다. 참고자료인 그림 8.4는 1982년 영국의 Beker가 조사하여 발표한 자료이다. 그리고 아래의 그림 8.5의 결과 화면을 보면 암호화 후에 a의 빈도가 가장 많은 것을 볼 수 있다. 따라서 우리는 e가 a로 치환되었다는 것을 알 수 있고, 곧바로 키의 값이 22라는 사실을 유추할 수 있을 것이다.

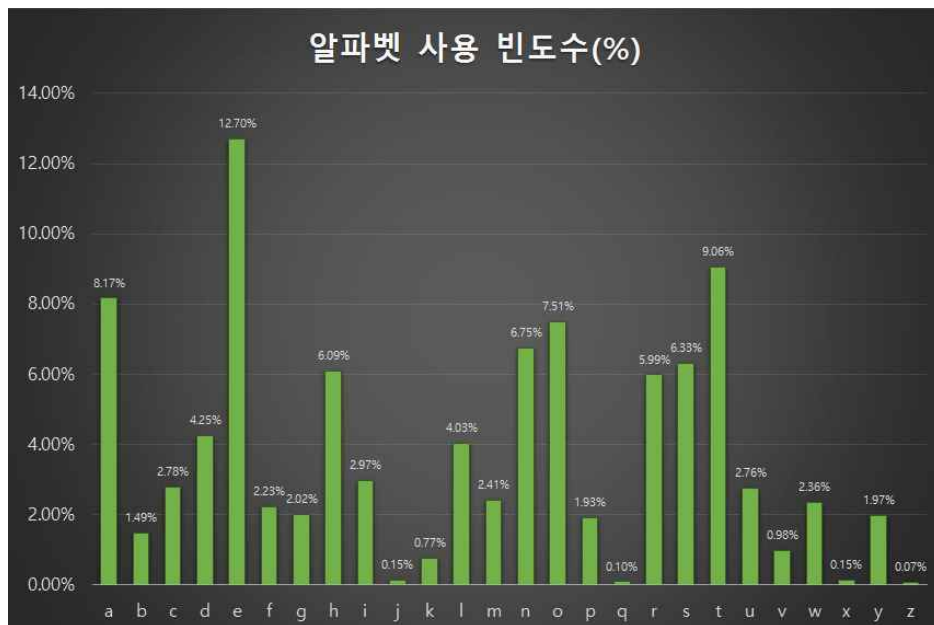


그림 8.4 알파벳 사용 빈도수(%)

| 암호화하기 전의 알파벳 사용 빈도수 | 암호화한 후의 알파벳 사용 빈도수 |
|--|--|
| <pre> C:\> C:\WINDOWS\system32\cmd.exe a : 7.714% b : 1.773% c : 2.617% d : 4.332% e : 13.124% f : 2.389% g : 1.969% h : 5.802% i : 7.188% j : 0.192% k : 0.608% l : 4.021% m : 2.571% n : 6.728% o : 7.219% p : 1.950% q : 0.104% r : 5.599% s : 6.059% t : 9.446% u : 3.203% v : 0.906% w : 2.239% x : 0.203% y : 1.969% z : 0.075% </pre> | <pre> C:\> C:\WINDOWS\system32\cmd.exe a : 13.124% b : 2.389% c : 1.969% d : 5.802% e : 7.188% f : 0.192% g : 0.608% h : 4.021% i : 2.571% j : 6.728% k : 7.219% l : 1.950% m : 0.104% n : 5.599% o : 6.059% p : 9.446% q : 3.203% r : 0.906% s : 2.239% t : 0.203% u : 1.969% v : 0.075% w : 7.714% x : 1.773% y : 2.617% z : 4.332% </pre> |

그림 8.5 알파벳 사용 빈도수