블록체인 기술 기반 보안용 생체인식 시스템에 관한 연구

이동석
한밭대학교 컴퓨터공학과

Biometrics security system based on blockchain technology

Dong-Seok, Lee
Dept. of Computer Engineering, Hanbat University

요 약

본 연구에서는 객체 지향 프로그래밍 언어인 JAVA를 기반으로 설계 및 컴파일된 생체정보 블록체인 시스템을 이용하여 신뢰성이 높은 정보보호 및 공유 환경을 구성하는 방법을 제안한다. 또한 제안된 연구는 향후 공격 가능성을 차단하기 위해 높은 연산량을 확보하는 클라우드 시스템 및 내부 네트워크 구성 또한 같이 서술한다.

1. 서론

2018년, 블록체인 기술 기반의 암호화폐에 관한 관심이 전례가 없을 정도로 많은 이목을 끌게 된다. 불과 10년 전까지만 해도 몇 달러 되지 않았던 비트코인은 현재 그 가치가 한화 5천만원 이상의 높은 가치를 지니게 되었으며, 그만큼 관련 기술에 관한 관심도 집중되었다.

비트코인은 암호화폐의 종류 중 하나로 가장 대중적이다. 암호화폐는 기본적으로 Hash 알고리즘을 기반으로 하는 데이터 위변조 방지 기술인 블록체인으로 설계된다.

불과 2000년대 초반까지만 해도, 정보보호는 실제로 서비스되는 영역에서 중요하게 여기는 문제가 아니었다. 사용자의 ID, 비밀번호는 어떠한 암호화도 거치지 않은 채 순수 날 것 그대로 저장되었으며 이마저도 DB에 저장되면 다행일 만큼 초기 정보보호와 그것의 구조는 정형화되지 않은 방법론의 집합이었다.

하지만 시간이 흘러 데이터의 홍수라는 말이 생긴 만큼 현실의 공간 속 객체들을 이진화시켜 저장하려는 시도 및 서비스들이 늘어났고 이에 따라 악의적인 목표를 가진 이용자들도 비례하여 많아졌다.

이러한 상황에 있어 정보보호에 관한 관심과 집중이 늘어났으며 현재에 이르러 대두 받는 기술인 블록체인에까지 내려져 오게 되었다.

블록체인은 Hash 알고리즘과 분산 컴퓨팅의 특성을 이용하여 정보의 무결성을 보호하는 시스템이며, 비단 암호화폐 분야뿐만 아닌 정보보호를 필요로 하는 모든 분야에서 사용될 수 있으며 이에 생체 정보도 예외는 아니다.

생체 정보는 집단이나 단체가 개인을 증명, 보증할 수 있는 일종의 신분증과 같은 의미를 지니며, 이러한 중요성에 따라 국가에서는 이를 국가보안 시설에서 관리한다.

본 연구에서는 블록체인의 특성과 안전성을 통해 생체 정보를 안전하게 관리하는 방법을 제안하며 추가적인 안전성을 위해 클라우드 시스템 및 내부 네트워크의 구성 또한 같이 서술한다.

2. 관련 연구

본 절에서는 해시, 블록체인, 생체 정보 등에 대해 알아본다.

2.1 해시

Hash Function이라고도 불리는 Hash Algorithm, 이하 해시는 특정 길이의 데이터를 고정된 길이의 데이터로 대응하는 함수를 말한다.

예를 들어, 어떤 정수든 해당 일의 자리 숫자를 반환하는 함수도 해시라 할 수 있다. 해당 함수는 43을 넣

었을 때 3을 반환하고, 1349534를 넣었을 때 4를 반환할 것이다.

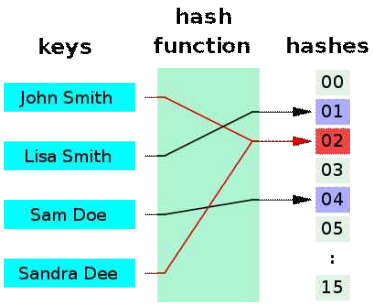

그림 1 해시의 작동원리

엄밀한 사전적 특성은 위와 같지만 본 논문에서 앞으로 서술할 해시는 아래와 같은 특성을 가진 것으로 한다.

1. 어떠한 입력값에도 항상 고정된 값을 출력한다.
2. 입력값이 조금만 바뀌어도 전혀 다른 값을 출력한다.
3. 출력된 값을 통해 입력값을 유추하는 것이 현재의 연산 능력으로는 불가능에 가깝다.

위의 조건을 충족하는 해시로는 대표적으로 SHA(Secure Hash Algorithm)가 있다. SHA는 해당 보안 레벨에 따라 1, 2, 3 버전이 있다.

일반적인 경우, 결괏값의 길이가 길수록 보안성을 요구하는 3번 특성이 고도화된다 볼 수 있다.

3번의 특성을 통해 우리는 정보보호에서 흔히 말하는 데이터 무결성(Integrity)을 충족시킬 수 있다. 데이터 무결성은 쉽게 말해 해당 데이터가 원본인지, 아니면 조작이 가해졌는지를 판단할 수 있는 척도라 할 수 있다.

비둘기 집의 원리에 따라, 다른 입력값에 같은 결괏값(해시값)이 나오는 상황은 무한히 존재하며, 이를 해시 충돌(Hash Conflict)이라고 부른다. 하지만 무한한 입력에 한해 무한한 해시 충돌이 존재하는 것이며 현실적인 기술력으로 볼 때, 해시 충돌을 일으키는 것은 그 가능성이 불가능에 가깝다.

SHA-1은 현재 기준 많은 집단에서 요구하는 해시 충돌 안전성을 충족하지 못하여 사장되었으며, 현재는 SHA-2, SHA-3를 많이 사용하는 실정이다.

비트코인은 SHA-2 중, SHA-256을 사용한다.

2.2 블록체인

블록체인은 일정 기록(블록)을 암호학을 통해 연결한 일종의 리스트이다. 이때 사용되는 알고리즘은 해시이며 그 결과물로 대개 아래와 같은 구조를 지닌다.

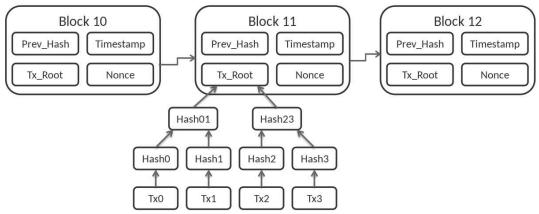

그림 2 블록체인의 구조

블록체인은 현재 생성된 블록의 해시값을 충족하는 입력값을 제시해야만, 즉 해시 충돌을 일으켜야만 다음 블록 생성이 가능해진다. 예를 들어, 특정 정숫값이 들어왔을 때 일의 자리 숫자를 반환하는 해시(14가 입력일 때 4가 출력인)가 모종의 블록체인이 채택한 해시라 해보자. 블록 A의 해시값이 6인 경우 다음 블록은 16, 26, 36등의 입력값들을 제시해야만 하는 것이다.

위의 사례와는 다르게 일반적인 해시 함수, 즉, 비트코인이 채택한 SHA-256 해시 함수의 경우 출력값으로 입력값을 추측할 수 없기에(다른 암호화폐 또한 그렇지만) 다음 블록을 만들기 위해서는 해시 충돌이 일어날 때까지 모든 경우의 수를 넣어보든 이른바, 무작위 대입 방식(Brute Force)을 사용할 수밖에 없으며 이는 매우 높은 연산량을 요구한다.

2.3 생체 정보

생체 정보(Biometrics)는 인간의 신체와 관련된 정보를 말한다. 생체인식(Recognition)과 높은 관련성을 지니는데, 생체 정보의 유사성을 통하여 개인을 식별하는 과정을 기치기 때문이다.

생체 정보로는 측정할 수 있는 모든 정보가 존재한다. 예를 들어 우리가 흔히 알고 있는 키, 몸무게, 시력, 혈압, 혈당 등이 그렇다.

하지만 개인 식별 목적으로 설계된 생체인식 시스템은 그 데이터로 쓸 수 있는 범위가 상당히 한정적인데 이것들에는 얼굴, DNA, 지문, 홍채, 망막 등이 있다. 이에 더하여 행동 특성으로 걸음걸이, 행동 분석, 필기체, 타자 습관 등이 추가로 존재한다.

비밀번호, 신분증, ID 카드와 같은 전통적인 접근 제한 방식은 생체인식을 통한 방식과는 상응하는 특성이 있다. 전통적 접근 제어 방식과 비교하여 생체인식을 위한 생체 정보는 개인마다 고유하고, 변형이 불가능에 가까우므로 더욱 안전하고 안정적이다. 하지만 사생활, 인도주의, 존엄성 등의 중요한 인륜적 가치문제 또한 동시에 안고 있는 실정이기에 현재로서는 정보보호 분야의 뜨거운 감자의 위치에 서 있다.

3. 블록체인 기술 기반 보안용 생체인식 시스템

본 절에서는 2. 관련 연구 절에서 서술한 내용을 기반으로 본 연구의 본제인 블록체인 기술 기반 보안용 생체인식 시스템의 필요성을 인식한 후 이를 설계 및 구현하며 시스템의 취약점과 대응 방안을 자세히 알아본 후 작동 결과를 확인한다.

3.1 필요성

2.3 단에서 서술하였듯, 생체 정보는 다른 전통적인 접근 제한 방식과는 다른 특성을 가지고 있다. 대체로 보았을 때, 인간 개인에 따라 1대 1 대응되는 고유한 값이며, 변형이 거의 불가(불변)하다는 점이 그렇다. 이것은 매우 중요한 특징이다.

불변이라는 특징은 장점이자 단점으로 작용한다. 한 번 정보가 노출된 경우 다시는 사용할 수 없다는 점이 그렇다.

데이터베이스 접근 방식이 지문인증이라고 가정할 경우, 악의적인 공격자 한 명이 관리자의 동선을 파악한 후, 이를 이용하여 지문을 탈취해 데이터화 시켜 데이터베이스에 접근할 수 있을 것이다.

이러한 경우를 만들지 않기 위해 현재 대부분의 생체 정보 사용은 접근 제어 시도가 물리적인 환경에 종속적이거나 해당 접근 수단이 폐쇄적인 경우, 전통적인 접근 제어 방식과 같이 사용되는 경우들이다. 데이터베이스에 접근하는 것이 아닌, 어떠한 장소를 들어가기 위해 사용하거나, 휴대폰의 지문인식 등이 그렇다.

이러한 본연적 문제를 해결할 방법은 아직까지 명확하게 제시되지 않았다. 불변의 특성은 대부분의 생체 정보를 사용하는 이상 따라오는 특징이기에 이를 인정하고 넘어가는 것이 중요한 것이다.

암호학은 발전했고, 안전한 정보보호 방법은 수없이 많다. 당장 AES256 암호화 방식으로 나의 개인정보, 민감정보, 비밀 등을 암호화하여 온라인에 업로드한다고 해도, 컴퓨터 공학계의 비약적인 발전이 존재하지 않는 이상 저자가 사망할 때까지 풀릴 일은 없을 것이다. 그럼에도 불구하고 왜 우리들의 비밀번호는 유출되고 계정이 탈취되는 일이 빈번한 것인가? 그것은 과정에서 일어나는 노출 및 취약점이 즐비하기 때문이다. 당장 특정 웹사이트에 로그인한다는 것만으로 암호가 한 번 노출된다는 것을 의미한다.

이를 안전하게 처리하기 위해서는 SSL, RSA 암호화 방식을 이용하여 세션 자체에 공개키 암호화 방식을 체결해야만 한다. 게다가 이러한 통신상 요소뿐만 아닌 저장 방식에서도 보안이 철저히 지켜져야 한다. 비밀번호와 같은 중요 정보들은 세션 간 암호화 이후, 원본을 확인 후, 해시를 이용하여 복호화가 불가능한 암호화를 진행하여 저장하는 경우가 대부분(클라이언트 단에서 해시화 하여 전송하는 예도 존재)이다. 하지만 이것마저도 매우 안전하지는 않은데, 악의적 공격자들이 Rainbow Table Attack(일반 사용자들이 많이 사용하는 비밀번호를 미리 해시화 시켜 표를 만든 후, 이를 계속 대입하여 접근 제한을 탈취하는 공격)을 이용할 수도 있기 때문이다. 이러한 공격을 막기 위해서 Salting(데이터를 그대로 해시하는 것이 아닌 문자열을 추가하는 등의 조작을 거친 이후 해시하는 것. 소금을 치는 것과 같다 하여 Salting이라 불린다)을 이용하여 다시 한번 방어한다. Salting의 경우에도 대부분 특정 문자열을 추가하는 방식으로 진행하는데 이러한 경우도 Rainbow Table에 존재할 수 있기에 원 데이터에 따라 동적인 Salting이 추천된다.

그림 3 KISA(한국인터넷진흥원)가 발표한 안전한 비밀번호 해시화 방법

위와 같이 관련 기술에 대한 충분한 이해와 체계적인 계획수립이 이어진 후, 한 치의 취약점도 존재하지 않게끔 파이프라인을 구성하여야만 우리의 정보는 안전하게 저장된다. 하지만 우리는 확신할 수 없다. 모든 서비스가 이러한 과정을 거쳐 내 정보를 암호화되지는 않을 것이다. 어딘가에서는 평문 그대로 저장돼있을 수 있다는 것이다. 그리고 일반 사용자는 대부분 비밀번호를 특정 패턴 속에서 생성하기 때문에 한 곳에서 비밀번호가 노출된 경우 대부분의 곳에서 비밀번호가 노출된 경우랑 다를 바가 없는 것이다.

생체 정보는 공개된 것이랑 다를 바가 없는 데이터라는 전제를 형성할 경우, 블록체인을 이용한 가상화폐들의 탈중앙화와 같은 속성을 가지게 된다. 암호화폐의 모든 거래 내역 공개는 모든 데이터 공개이며, 이는 모든 생체 정보 공개로 치환될 수 있다. 전통적 접근 제어 수단은 추가적인 무결성 보증 암호화(해시와 같은)가 이루어져야 블록에 올릴 수 있지만 생체 정보는 그럴 필요가 없다.

물론 저자가 주장하는 방식은, 현재로서는 식별되지 않은 수많은 보안 위험성을 초래할 수 있으며 기존에 적용된 시스템들에 치명적인 결함을 일으킬 수 있다. 따라서 본 연구에서는 기존 시스템의 방식 또한 수용하기 위해 추가적인 암호화를 거친 생체 정보를 블록체인에 담는 방법도 같이 진행한다.

3.2 설계

본 연구에서는 객체 지향 언어인 JAVA를 기반으로 블록체인 시스템을 직접 설계한다. 또한 해당 시스템을 테스트할 Docker(가상화 시스템) 및 GCP(Google Cloud Platform) 또한 추가적으로 설계한다.

아래는 JAVA로 설계되는 블록체인 기술 기반 보안용 생체인식 시스템이 가져야 할 요소들을 나열한다.

1. 블록의 이전 해시값 증명
- 그림 2에서 볼 수 있듯, 블록은 해시를 기반으로 서로가 Linked List의 형태로 이어지기 때문에 이를 반영해야만 한다. Block이라 부를 수 있는 Class는 이전 Block 해시값을 증명할 데이터를 소유하고 있어야 한다.

2. 현 블록의 해시값
- 데이터를 담고 있는 블록의 해시값이다. 현재 많은 암호화폐는 Nonce 등을 통해 블록의 해시값을 블록이 생성될 때 명시적으로 작성하지 않지만, 그것은 고도의 블록체인 시스템을 만들 때 필요한 기술이기에 본 연구에서는 블록이 생성될 때 블록 내부에 위치한 데이터들을 SHA-256 함수에 넣어 나오는 해시값을 블록의 해시값으로 하기로 한다.

3. 블록의 생체 정보 목록
- 블록의 주목적이라고 할 수 있는 생체 정보들의 목록이다. 블록에 담기는 생체 정보들은 블록이 생성되는 시점까지 수신받은 정보들의 집합이라 할 수 있다. 본 연구에서 사용할 생체 정보는 지문과 기본적인 얼굴 인식에 필요한 테두리 데이터이다.

4. 블록의 생성 시각
- 블록의 생성 시각은 단순한 Date 형 자료이지만, 블록의 해시값에 영향을 미친다.

5. 블록의 난이도
- 현재의 기술력으로 SHA-256 전문의 충돌을 일으키는 것은 불가능에 가깝다. 따라서 SHA-256으로 암호화를 진행하되, 블록의 생성 시간의 차이를 계산하여 난이도를 조정, 특정 부분만 일치해도 블록 생성이 가능하게끔 설정한다.

6. 네트워크 모듈
- 블록체인 시스템에 서버는 필요로 하지 않지만, 국소적인 테스트 환경에서는 이를 생성하기로 한다. 서버와 클라이언트의 통신 규약을 TCP로 정하고, JAVA Object를 직렬화하여 통신하는 것으로 결정한다.

7. Dockerfile

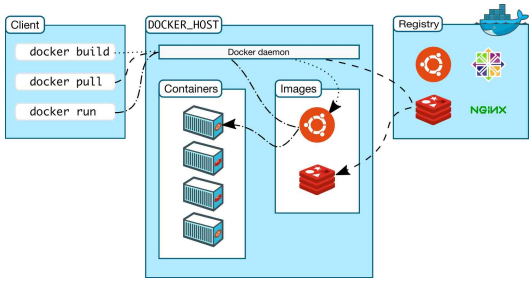

그림 4 Docker의 아키텍처

- Docker는 개인용 PC에서 블록체인에 필요한 다수의 노드와 내부 네트워크 환경을 실험적으로 구성할 수 있게끔 보조하는 가상화(Virtualization) 도구이다. 이러한 Docker가 올바르게 작동할 수 있게끔 하는 명세서인 Dockerfile의 작성이 필요하다.

3.3 구현

3.2 설계 단에서 나열한 요소들의 구현체들을 서술한다. 블록과 관련된 내용은 대부분 연구의 형식을 위하여 간소화되었으며 전문은 본 논문의 각주에서 확인할 수 있다.

1. 블록의 이전 해시값 증명
- Block이라 부를 수 있는 Class는 이전 Block 해시값을 증명할 데이터를 소유하고 있어야 한다. 또한 해당 값은 컴퓨터가 직접 찾아내야 하며 이를 위해 아래와 같이 무작위 값을 넣어 해시값을 찾는 Mining(채굴)이라 하는 과정을 거치도록 코드를 구현한다.

```
public static String getHashConflict(String target)
{
    StringBuilder sb = new StringBuilder();
    while(true)
    {
        if (sb.length() > 20) sb.setLength(0);
        sb.append(getRandom());
        String ar = sha256(sb.toString());
        assert ar != null;
        ar = ar.substring(ar.length() - target.length(), ar.length());
        if (ar.equals(target)) return sb.toString();
    }
}
```

표 1 해시 충돌을 찾는 함수

2. 현 블록의 해시값

- 블록이 소유한 생체 정보 객체인 BioInfo 객체를 모두 해시화 시켜 해당 값들을 모두 더한 후, 현재 날짜 및 이전 블록의 해시값을 추가하여 연산을 수행한다.

```
public String getContentHash()
{
    StringBuilder cntHash = new StringBuilder();
    for(BioInfo bi : contents)
    {
        cntHash.append(bi.getHash());
    }
    return Hash.sha256(preBlockHash + Long.toString(timestamp) + cntHash.toString());
}
```

표 2 블록의 해시를 만드는 함수

3. 블록의 생체 정보 목록

- 생체 정보 클래스는 BioInfo 라는 부모 클래스를 토대로 FaceBioInfo, FingerprintBioInfo 등의 자식 클래스들이 존재한다.

```
public class BioInfo implements Serializable
{
    protected String hash;
    protected Boolean isEncrypted;
    protected BioInfoType type;
    protected String encryptedData;

    // for Serialize
    public BioInfo() {
    }

    public BioInfo(Boolean isEncrypted, BioInfoType type, String encryptedData)
    {
        this.isEncrypted = isEncrypted;
        this.type = type;
        this.encryptedData = encryptedData;
    }

    public String getHash() {
        return hash;
    }

    public void setHash(String hash) {
        this.hash = hash;
    }

    public Boolean getEncrypted() {
        return isEncrypted;
    }

    public void setEncrypted(Boolean encrypted) {
        isEncrypted = encrypted;
    }

    public BioInfoType getType() {
        return type;
    }

    public void setType(BioInfoType type) {
        this.type = type;
    }

    public String getEncryptedData() {
        return encryptedData;
    }

    public void setEncryptedData(String encryptedData) {
        this.encryptedData = encryptedData;
    }
}
```

표 3 자식 클래스의 부모가 되는 BioInfo 클래스의 전신

4. 블록의 생성 시각

- 단순한 구현이기에 생략한다.

5. 블록의 난이도

- 2. 현 블록의 해시값 구현에서 서술하였으므로 생략한다.

6. 네트워크 모듈

- 서버와 클라이언트는 JAVA Object를 직렬화하여 TCP 로 통신한다.

```
public class Server
{
    int SOCKET_PORT = 3377;
    List<Block> chain = new ArrayList<>();

    public void start()
    {
        Block firstBlock = new Block();
        firstBlock.setHash("9");
```

```
            chain.add(firstBlock);
            System.out.println("ORIGIN BLOCK CREATED(HASH : " + firstBlock.getHash() + "), WAITING FOR NEXT BLOCK");

            try
            {
                ServerSocket ss = new ServerSocket(SOCKET_PORT);
                Socket cl = null;
                PrintWriter pw;
                ObjectInputStream ois;

                while(true)
                {
                    cl = ss.accept();
                    //System.out.println("SOCKET ACCEPTED");
                    pw = new PrintWriter(new BufferedWriter(new OutputStreamWriter(cl.getOutputStream())));
                    ois = new ObjectInputStream(cl.getInputStream());

                    TransBlock tb = (TransBlock) ois.readObject();

                    switch (tb.transType)
                    {
                        case ADD_BLOCK:
                            try
                            {
                                String answerHash = Hash.sha256(tb.getBlock().getPreHashAnswer());
                                String preHash = chain.get(chain.size() - 1).getHash();
                                String answerHashFit = answerHash.substring(answerHash.length() - preHash.length(), answerHash.length());
                                if (answerHashFit.equals(preHash))
                                {

                                    chain.add(tb.getBlock());

                                    System.out.println("BLOCK(" + (chain.size() - 1) + ") APPLIED(ANSWER : " + tb.getBlock().getPreHashAnswer()+", HASH : " + tb.getBlock().getHash() + ")");
                                }
                                else
                                {

                                    System.out.println("BLOCK FAILED");
                                }
                            }
                            catch(Exception e)
                            {
                                e.printStackTrace();

                                System.out.println("BLOCK FAILED");
                            }
                            break;
                        case GET_BLOCK:
                            ObjectOutputStream oos = new ObjectOutputStream(cl.getOutputStream());
                            oos.writeObject(chain);
                            oos.flush();
                            break;
                    }

                    pw.close();
                    cl.close();
                }
            }
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }

        public static void main(String[] args)
        {
            Server sv = new Server();
            sv.start();
        }
}
```

표 4 TCP 통신을 통한 TransBlock의 수신부

7. Dockerfile

- Ubuntu 20.04 기반 Dockerfile을 작성하였으며 Java는 8, 11버전을 각각 설치하였다.

```
# LEE DONGSEOK(leeds06080@gmail.com)

# VERSION : 5
# docker build --build-arg USERID=userid --build-arg USERPWD=userpwd -t ubu:r5 .
# docker run -d ubu:r5

# FROM ubuntu:latest
FROM ubuntu:20.04

ARG USERID
ARG USERPWD

RUN apt update
RUN apt -y install sudo

RUN useradd -ms /bin/bash $USERID
RUN printf "$USERID:$USERPWD" | chpasswd
RUN usermod -aG sudo $USERID
RUN printf "$USERPWD\n" >> /password
RUN chown $USERID /password
RUN chmod 700 /password

RUN apt install -y htop
RUN apt install -y curl
RUN apt install -y wget
RUN apt install -y nano
RUN apt install -y net-tools
RUN apt install -y iputils-ping
RUN apt install -y language-pack-ko
RUN apt install -y openssh-server
RUN apt install -y lrzsz

# python is installed since ubuntu 20.04
RUN apt install -y python3-pip

# starter
RUN printf "cat /password | sudo -S service ssh start\nexport LC_ALL=ko_KR.UTF-8\n\necho \"SUCCESSFULLY STARTED\"\n\n# DELETE LINES BELOW AFTER DONE WRITING THIS FILE\nwhile true; do sleep 1; done\n" >> /starter.sh
RUN chown $USERID /starter.sh

EXPOSE 22
USER $USERID
ENTRYPOINT ["/bin/bash", "/starter.sh"]
```

표 5 기본적인 Ubuntu의 Dockerfile

```
# LEE DONGSEOK(leeds06080@gmail.com)

# VERSION : 3
# first version is ubu image's
# docker build --build-arg USERID=userid -t ubuj:r5r3 .

FROM ubu:r5

ARG USERID

RUN cat /password | sudo -S apt update

# java
RUN cat /password | sudo -S apt install -y openjdk-8-jdk
RUN cat /password | sudo -S apt install -y openjdk-11-jdk
RUN cat /password | sudo -S mkdir /jbase
RUN cat /password | sudo -S chown -R $USERID:$USERID /jbase
RUN cat /password | sudo -S chmod -R 755 /jbase
RUN mkdir /jbase/jars
RUN printf "java -jar /jbase/jars/*.jar" >> /jbase/starter.sh
```

표 6 Java 설치용 Dockerfile

Dockerfile 서술을 마친 이후 Image화 시켜 아래와 같이 컨테이너로 등록하였다.

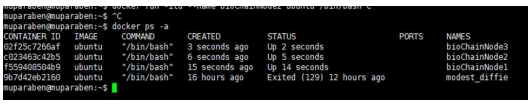

그림 5 컨테이너로 등록된 Docker 이미지

3.4 취약점

3.2 설계 및 3.3 구현에서 서술된 시스템은 블록체인 기반이기에 가질 수밖에 없는 한계가 명확히 존재한다.

공격자가 악의적인 의도를 갖고 높은 연산력을 기반으로 한 공격이 들어올 경우, 50%의 점유율 이상을 차지한 시점에서 블록체인 시스템은 점유권 및 안정성을 보증할 수 없게 된다.

따라서 본 연구에서는 이러한 연산력 부족을 해결하기 위해 클라우드 시스템을 제안한다.

본 연구에서는 여러 개의 클라우드 서비스 중 하나를 선택하기 위하여 아래와 같은 기준을 적용했다.

- 보안성
- 독립 네트워크 환경 구성 용의성

- 확장성
- 제어 용의성

이 네 사항과 관련하여 Amazon의 EC2와 Google의 Google Cloud Platform, 이하 GCP가 조사되었고 이 중 여러 요소를 추가로 고려하여 최종적으로 GCP로 선택하게 되었다.

3.4 취약점 대응

GCP는 Python 라이브러리 중 libcloud 로 조작이 가능하다.

```
from libcloud.compute.types import Provider
from libcloud.compute.providers import get_driver
from libcloud.compute.types import NodeState

def getDriver(id:str, pwdPath:str, projectName: str):
    ce = get_driver(Provider.GCE)
    return ce(id, pwdPath, project=projectName)

def getNode(driver: str, name: str):
    return driver.ex_get_node(name)

def getNodeState(driver: str, name: str):
    return str(getNode(driver, name).state)
```

표 7 libcloud 사용 코드의 일부

본 연구에서는 해당 라이브러리를 이용하여 블록의 생성 시간이 짧아질수록, 그리고 식별 불가능한 IP가 네트워크에 참여할수록 GCP의 인스턴스를 생성하여 안전성이 높은 연산력을 확보할 수 있도록 유도했다.

GCP 인스턴스는 asia-northeast3(서울) 리전의 asia-northeast3-a 영역을 선택하였으며 머신 구성은 일반 용도로 선택했다.

테스트를 위해 전문적이고 고도화된 지식이 많이 필요한 타 시리즈보다는, 일반적이고 범용적인 연산에 적합한 E2 시리즈를 골랐다. 머신 유형은 e2-medium으로 Virtual CPU가 2개이며, 4GB 메모리가 탑재되어 있다.

해당 인스턴스는 상기 서술된 Docker Container와 동일한 환경인 Ubuntu 20.04에서 생성되었다.

생성이 완료되었을 때 월 32달러, 시간당 0.04달러로 저렴한 유지 비용을 확인할 수 있었다.

그림 6 GCP에 등록된 인스턴스들

그림 6에서 확인할 수 있듯, 각각의 인스턴스들은 상호 간에만 통신할 수 있는 내부 IP 대역을 갖추었으며 이에 따라, 관리자가 원할 경우 외부 IP를 차단하고 내부적으로만 통신을 수행할 수 있다.

이러한 폐쇄망 접근은 블록체인 네트워크의 Private Network라 볼 수 있으며, 민감할 수 있는 생체 정보를 인가된 사용자 또는 집단만 이용하게끔 할 수 있다. 외부 접근을 효율적으로 차단하기 위하여 GCP 내부적으로 네트워크를 수립하거나, 외부 서비스에서 VPN, V-LAN 등을 만들어 적용하는 방법 또한 고려해볼 만하다.

3.5 작동

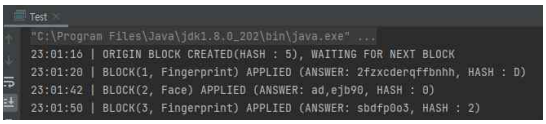

그림 7 작동 로그

그림 7에서 확인할 수 있듯, 시스템을 처음 가동할 경우, 설정한 난이도에 따른 랜덤한 해시값을 공개하고 블록 생성을 기다린다.

새로운 블록을 만든 경우 클라이언트는 서버에 그에 따른 ANSWER 값과 자신의 해시값을 데이터와 함께 전송하는 모습을 확인할 수 있으며 이에 서버는 해당 데이터가 어떠한 종류의 생체 정보인지 로그로 저장한다.

이전 블록의 해시값은 다시 새로운 블록을 만들려는 노드에서 해시 충돌로 유도해 내어야 할 값이 되며 이로 인하여 그림 7에서 계속해서 달라지는 해시 입력값들을 확인할 수 있다.

위의 모든 과정은 내부 네트워크 환경에서 진행되기에 외부로부터 해당 데이터의 내용을 식별할 수 있는 수단은 존재하지 않는다. 이는 Docker Container간 통신과, GCP 내부 네트워크 통신 두 방향에서 모두 확인했다.

4. 결론

생체 정보는 하나의 디지털 보안 정보로, 그것이 공개되어도 되는지에 관한 의견은 계속하여 찬반이 갈라지고 있다.

하지만 그러한 생체 정보의 특징에도 불구하고 블록체인 시스템에 효율적으로 탑재할 수 있다는 점을 본 연구가 보여주고 있다.

50% 공격에 대한 보안 취약점 측면이 어느 정도 존재한다는 점은 사실이다. 다만 상기 서술했듯, 기술 발전을 통한 Cloud 시스템을 적극적으로 활용하면 이러한 현상 또한 큰 어려움 없이 극복할 수 있을 것으로 보인다.

이렇듯, 블록체인은 암호화폐에 국한되는 분산 컴퓨팅 기술이 아니기에, 비단 생체인식을 위한 생체 정보 보호뿐만이 아닌 여러 분야 및 데이터를 전파 및 보호하는 형식으로 응용될 수 있다는 것을 본 연구를 통해 말하는 바이다.

5. 참고 문헌

〔1〕 Narayanan, Arvind; Bonneau, Joseph; Felten, Edward; Miller, Andrew; Goldfeder, Steven (2016). Bitcoin and cryptocurrency technologies: a comprehensive introduction. Princeton: Princeton University Press. ISBN 978-0-691-17169-2.

〔2〕 https://github.com/Acet-Aminophen/bio_blockchain

〔3〕 https://github.com/Acet-Aminophen/Dockerfiles

[4] Knuth, D. 1973, The Art of Computer Programming, Vol. 3, Sorting and Searching, p.527. Addison-Wesley, Reading, MA., United States

〔5〕 Villas-Boas, Antonio. "Passwords are incredibly insecure, so websites and apps are quietly tracking your mouse movements and smartphone swipes without you knowing to make sure it's really you". Business Insider. Retrieved 22 November 2021.

[6] Bheemaiah, Kariappa (January 2015). "Block Chain 2.0: The Renaissance of Money". Wired. Archived from the original on 14 November 2016. Retrieved 13 November 2016.

이동석(Dong Seok, Lee) 한밭대학교 컴퓨터공학과

2018년 2월 : 대전 서일고등학교 졸업

2018년 3월 : 한밭대학교 컴퓨터공학과 입학

2020년 5월 : ㈜아이와즈 입사

2022년 2월 : 한밭대학교 컴퓨터공학과 졸업 예정

<관심분야> 정보보호

지도교수님 확인 : (인)