재구성 가능한 보안 모듈을 이용한 역컴파일 방지 Java 실행 환경

김성은O 유혁
고려대학교 컴퓨터학과
{sekimO, hxy}@os.korea.ac.kr

Decompilation-Preventive Java Runtime Environment by Dynamic Reconfigurable Security Module

SungEun KimO Hyuck Yoo
Dept. of Computer Science & Engineering, Korea University

요 약

현재의 네트워크화 된 환경에서 Reverse Engineering 방지의 중요성은 높아지고 있다. 특히 Java의 경우 Reverse Engineering에 의한 위험성에 더욱 노출되어 있다. 본 논문에서는 이러한 환경에서 Reverse Engineering 방지에 대한 기존의 연구에 대하여 검토하고, Java 바이트 코드의 암호화를 통한 방어 기법으로서 동적 인스트루멘테이션을 이용한 재구성이 가능한 새로운 Java 실행 환경을 제안하였다. 이를 통하여 보다 신뢰성 있는 모바일 에이전트 환경을 구축할 수 있을 것이다.

1. 서 론

기존 네트워크의 영역이 확장되고, 모바일 기기의 인터넷 접속에 대한 가능성이 높아지면서 모바일 에이전트의 중요성이 확장되고 있다. 이와 더불어 Java의 출현은 많은 모바일 에이전트 플랫폼의 개발을 이끌었다. Java는 모바일 에이전트에 필요한 많은 기능성을 내장하고 있다. Java 런타임 시스템은 거의 모든 하드웨어 플랫폼 및 운영체제 상에서 동작 가능하고 상위 어플리케이션에 동일한 환경을 제공한다. 그러므로 Java 위에서 동작하는 모바일 에이전트 플랫폼은 높은 이식 가능성을 보장하고 있다. 모바일 코드는 필수적으로 아키텍처에 독립적인 포맷으로 작성되어져야 한다. 이로 인하여 많은 모바일 에이전트 시스템이 Java 환경에서 개발되고 사용되어지고 있다[1][5].

이러한 환경에서 소프트웨어는 과거에 비해 보다 더 분산적이고 아키텍처 독립적으로 존재하게 되었다. 분산적인 환경 하에서 소프트웨어 보호의 중요성은 강조되고 있다. 소프트웨어는 그 자체로는 데이터의 형태를 가지고 있다. 데이터로서의 소프트웨어에 대한 조작 가능성은 항상 존재하며, 조작을 통한 새롭게 만들어진 소프트웨어의 오용은 심각한 문제로 대두되어지고 있다[2]. 특히 Java의 경우 Reverse Engineering에 의한 위험성에 더욱 노출되어 있다. 일반적으로 C로 쓰여지고 컴파일된 오브젝트 코드는 “ strip”의 과정을 거치면 Reverse Engineering의 가능성이 크게 줄어든다. 오브젝트 코드에 대한 “ strip”은 오브젝트 코드에서 변수 이름이나 라이브러리 루틴에 대한 정보를 제거한다. 예를 들면 C 언어 라이브러리 루틴 중 “ printf”에 대한 정보는 strip된 오브젝트 코드에서 특정 메모리 주소에 대한 프로시저 콜로 저장되어진다. 그러나, Java는 플랫폼에 독립적인 바이트 코드 형태로 컴파일이 이루어진다. Java 바이트 코드는 strip된 C 오브텍트 코드에 비하여 소스 코드에 대한 많은 정보를 포함하고 있기 때문에 상대적으로 역컴파일이 용이하다. 실제 현재 Java 바이트 코드에 대한 역컴파일러는 매우 많이 이용되고 있으며, 그 성능도 뛰어나다[3][4]. 그러므로, Java 바이트 코드에 대한 Reverse Engineering의 문제는 타 언어를 이용한 시스템에 비해 더욱 중요하다. 다른 말로 서술하면, 지적 자산으로서의 모바일 에이전트 소스 코드를 악의적인 그룹이 불법적으로 취득할 가능성이 타 언어에 비해 높다고 할 수 있다.

본 논문은 Java 바이트 코드에 대한 Reverse Engineering을 방지하는 기존의 방법을 검토하고, 효과적인 방지 방법을 제시하는데 그 목표를 둔다. 이를 위해 Java 실행 환경에 동적으로 재구성 가능한 Security 모듈에 대하여 제시하고, 이를 이용하여 Java 바이트 코드에 대한 Encryption/Decryption을 수행하여 Reverse Engineering에 대한 취약한 Java 환경을 개선하는 방법을 제시한다.

2. 사례 : Reverse Engineering이 파생하는 문제점들

소프트웨어에 대한 불법적인 Reverse Engineering을 방지하기 위해 개발자는 프로그램 내부에 보호 메커니즘을 구현한다. 그러나, 불법적인 이용자는 이러한 보호 메커니즘을 구현한 코드를 프로그램 내부에서 찾아내서 제거할 수

있다.

앞에서 서술한대로 현재의 기기들은 네트워크에 연결되어 사용되는 추세이다. 예를 들어 개인용 미디어 재생기나 셋톱 박스의 경우 소프트웨어에 적용되어 있는 이용 정책에 의하여 디지털 컨텐츠를 다운로딩하고 재생할 수 있다. 디지털 컨텐츠는 프로그램 내부에 포함되어 있는 Encryption 및 Decryption, 디지털 서명 및 서명 검증, 공인키 등에 대한 기본 기능 및 관련 인프라 구조를 통하여 보호될 수 있다. 그러나, 이용 정책이 적용되어 있는 소프트웨어에 대한 불법적인 Reverse Engineering이 이루어져 이용 정책에 대한 악의적인 수정이 가해질 경우, 요금의 지불 없는 영화의 다운로딩, 다수의 시청, 그리고 디지털 컨텐츠의 권리 보호 기능이 제거된 디지털 복사본이 만들어질 수 있다.

이러하듯이 Reverse Engineering에 의한 소프트웨어의 악의적인 수정 가능성은 단순한 지적 자산으로서의 프로그램 소스코드의 공개라는 측면 뿐만 아니라, 모바일 에이전트 환경에서의 디지털 컨텐츠의 정당한 과금 기반의 유통 구조를 왜곡시켜 디지털 컨텐츠의 권리를 소유한 집단에 손실을 유발할 수 있다는 측면을 고려하여 사고되어져야 한다. 소프트웨어 제공자는 신뢰성있고 안정한 소프트웨어을 제공함에도 불구하고, 이용자의 기기에 설치되고 난 이후의 소프트웨어는 이러한 Reverse Engineering에 의한 수정 가능성에서 자유로울 수 없다.

명백하게, 이러한 위험에서 소프트웨어를 보호하기 위한 보다 효율적이고 효과적인 메커니즘 개발의 요구는 존재한다. 그러나, 모바일 에이전트가 동작하는 “Open Computing Platform” 상에서 현재까지 제안된 어떠한 보호 메커니즘도 완벽하게 소프트웨어를 보호하고 있지 못하다[6]. 물론 “Closed Box” 에서는 효과적인 보호 메커니즘이 존재하지만, 이는 모바일 에이전트의 기본 환경과는 그 성격이 맞지 않는다. 그러므로, 모바일 에이전트의 기본 환경으로서의 “Open Computing Platform”를 고려할 때, 불완전함에도 불구하고 소프트웨어 보호 메커니즘의 개발은 여전히 필요하다. 우리가 Reverse Engineering에 대한 보호를 하는 것이 불가능하게 하다고 한다면, 우리는 보호 메커니즘의 적용을 통하여 적어도 악의적인 이용자의 Reverse Engineering에 필요한 시간과 노력을 증가시킬 수는 있을 것이다.

3. Reverse Engineering 방지 메커니즘

Java로 구현된 소프트웨어에 대한 불법적인 Reverse Engineering을 방지하기 위해 제시된 메커니즘으로 클라이언트-서버 모델, Native 오브젝트 코드의 전송, 코드 Obfuscation, 암호화된 바이트 코드 등이 있다. 이 절에서는 이러한 메커니즘들의 장점 및 단점에 대하여 서술한다.

3.1. 클라이언트-서버 모델

소프트웨어에 대한 Reverse Engineering의 가능성을 제거하는 가장 급진적인 방식은 프로그램을 믿을 수 있는 시스템 내부에서만 동작하도록 하여 프로그램에 대한 물리적인 접근을 막는 것이다. 즉, 소프트웨어 제공자의 서버에서 프로그램 대부분의 동작을 담당하고, 이용자와는 제한된 숫자의 서비스 인터페이스를 통하여 통신하도록 하여 이용자와 소프트웨어를 물리적으로 격리 시키는 방식이다[7]. 예를 들어 HTML이나 XML의 경우 실행 코드는 웹 도큐먼트에서 스크립트의 형태로 포함되어 있다. 이러한 방식은 웹 컨텐츠의 전송에 널리 쓰이는 방식이나, 그 자체의 확장성의 문제를 지니고 있고, 네트워크 대역폭의 제약으로 인한 성능 저하의 문제를 지니고 있다.

3.2. Native 오브젝트 코드

앞에서 기술한 바와 같이 C 오브젝트 코드는 Java 바이트 코드에 비해 Reverse Engineering에 대한 취약성이 현저하게 적다. 그러므로, Java 바이트 코드 대신 네이티브 오브젝트 코드로 제공하는 방식은 악의적인 이용자의 역컴파일에 대한 노력과 비용을 증가시켜 소프트웨어에 대한 조작 가능성을 줄일 수 있다. 그러나, 네이티브 오브젝트 코드는 Java 가상 머신에서 제공하는 Type-Safe 검증 기능을 지원하고 있지 않다. 그러므로, Buffer-Overflow 공격이나 바이러스와 같은 악의적인 코드에 대한 취약성이 반대로 증가한다. 이러한 취약성은 디지털 서명을 이용하여 C 오브젝트 코드의 신뢰성을 보증하여 완화시킬 수 있으나, 서로 다른 아키텍처에 서로 다른 버전이 필요하다[8]. 이는 소프트웨어 유지에 필요한 노력이 증가함을 의미한다. Java의 경우 가상 머신 상에서 동작하기 때문에 단지 하나의 버전만 필요하다. Java 바이트 코드의 Native 오브젝트 코드로의 변환 방식의 또 다른 단점으로는 코드 크기를 들 수 있다. “Hello, World!”를 구현한 간단한 Java 프로그램을 Native 오브젝트 코드로 변환시킨 경우 코드의 크기가 수십 MByte에 달한다. 이는 제한된 네트워크 자원을 이용하는 모바일 코드에 있어서는 치명적인 단점으로 작용한다.

3.3. 코드 Obfuscation

코드 Obfuscation은 프로그램 고유의 기능성은 유지하면서 프로그램을 이해하기 어려운 방식으로 변환하는 것이다[8]. 이러한 방식에는 식별자의 변환을 통한 Layout Obfuscation, Data의 값에 대한 변환을 통한 Data Obfuscation, Control Flow의 조작을 통한 Control Obfuscation 등이 있다. 이는 프로그램 성능의 저하 및 코드 추가에 의한 예기치 못한 부작용 등에도 불구하고, 해당 프로그램은 동일한 결과를 보장한다. 이러한 성능 저하의 측면은 코드의 안전성과 Trade-off 관계로 고려되어진다.

코드 Obfuscation은 기본적으로 완벽한 Reverse Engineering에 대한 방어를 보장하지는 않는다. 대신, 악의적인 이용자의 Reverse Engineering에 필요한 노력과 비용을 증대시켜 Reverse Engineering을 비경제적인 행위로 만드는데 그 목적이 있다. 이러한 목적은 악의적인 이용자가 해당 코드에 대한 이해를 어렵게 하여 그 비용이 도일한 기능을 지닌 새로운 프로그램을 작성하는 것 보다 많이 들도록 강제하도록 한다는 면에서 타당하다.

3.4. 암호화된 바이트 코드

코드에 대한 수정을 막는 가장 확실한 방법은 이용자가 코드의 내부를 볼 수 없게 하는 것이다. 코드에 대한 암호화

는 이용자의 코드 내부에 대한 접근을 차단하는 방법으로 이용된다[9]. 이 방법은 직관적인 설득력을 지니고 있으나, 몇 가지 문제점을 지니고 있다. 첫째, 신뢰성 있는 인프라 구조가 지원되지 않으면 암호 키를 배포하는데 있어 문제가 발생할 수 있고 이용자는 키를 취득하여 암호화된 코드를 해독할 수 있다. 둘째, 암호화된 코드는 실행 전 해독을 수행해야 하는데, 이는 필연적으로 성능의 저하를 가져온다. 셋째, 가장 중요한 문제로 암호가 해제된 코드에 대하여 이용자가 접근을 차단하고, 그 코드에 대하여 저장이 불가능하여야 한다. 이러한 문제를 해결하기 위하여 TCG에서 제안하는 TPM[10] 등의 하드웨어적인 방법을 이용할 수 있으나, 이는 프로그램의 이식성을 약화시키는 단점을 가진다.

4. 역컴파일 방지 Java 실행 환경

본 절에서는 Java 바이트 코드에 대한 Reverse Engineering을 방지하는 메커니즘으로 Java 바이트 코드에 대한 암호화 기법을 제안한다. 먼저 이러한 암호화 기법에 대한 기본적인 설계 원칙을 제시하고, 이를 적용하여 새롭게 구성한 Java 실행 환경에 대한 구조를 제안한다.

4.1. 암호화 기법 설계 원칙

코드에 대한 암호화 방식을 이용한 기존 연구의 단점을 해결하기 위해 다음의 설계 원칙을 도입한다.

1. 암호 키 및 이를 이용한 암호화된 코드 해독 과정은 이용자의 접근에서 격리되어야 한다.
2. 암호가 해제된 코드는 이용자의 기기에 저장될 수 없어야 한다.
3. 이식성을 손상시키지 않도록 코드 보호 메커니즘에서 부가적인 하드웨어의 사용을 배제하여야 한다.
4. 실행 환경은 모바일 코드를 안전하게 실행할 수 있는 Java 가상 머신 상으로 한정한다.

이 중 1과 2를 중심으로 고려하여 역컴파일 방지 Java 실행 환경을 구성하였다.

4.2 역컴파일 방지 Java 실행 환경

위의 설계 원칙에 기반한 Java 실행 환경은 [그림 1]과 같이 소프트웨어 제공자와 이용자 사이의 통신을 통하여 이루어진다. 이용자는 자신의 이용자 정보 및 플래폼에 관한 정보를 담고 있는 Identification을 소프트웨어 제공자에 전달하고, 소프트웨어 제공자는 이 Identification을 확인하여 이용자에게 암호화된 Java 바이트 코드(xxx.crypt) 및 이를 해독하여 정상적인 Java 바이트 코드를 생성할 수 있는 Decrypt 모듈을 전송한다. 여기서 Decrypt 모듈은 Native 오브젝트 코드의 형태로 작성되며, Java 실행 환경의 클래스 로더에 동적으로 배치될 수 있도록 클래스 로더와 인터페이스를 고려하여 구성한다.
Decrypt 모듈을 Native 오브젝트 코드의 형태로 작성하여 전송하는 이유는 앞서 기술한 설계원칙 1을 지키기 위함이

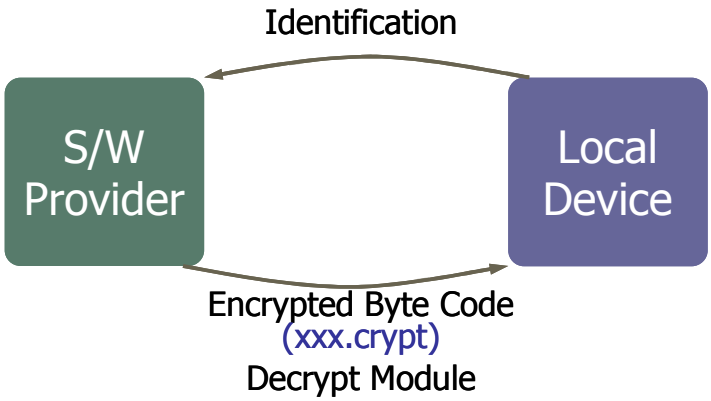

그림 1 소프트웨어 제공자와 이용자 간의 통신

다. 만약 Decrypt 모듈을 Native 오브젝트 코드가 아닌 Java 바이트 코드로 구성한다면, Decrypt 모듈에 대한 Reverse Engineering을 통하여 코드 해독 과정에 대한 이용자의 접근이 가능해지며, 코드 해독 과정의 분석을 통하여 제공되어진 암호화된 Java 바이트 코드에 대한 역컴파일이 가능하게 되어, 코드에 대한 보호가 실패할 수 있다. 이를 방지하기 위하여 Decrypt 모듈을 Java 바이트 코드에 비하여 Reverse Engineering이 상대적으로 어려운 Native 오브젝트 코드의 형태로 구성한다. 그리고, 암호 키 대신 Decrypt 모듈의 형태를 취하는 이유는 앞의 이유와 마찬가지로, 설계 원칙 1을 지키기 위함이다. 암호 키의 형태로 전송할 경우 이용자는 암호 키가 들어있는 네트워크 패킷의 데이터를 잡아내어 암호 키를 얻어낼 수 있고, 이를 이용하여 Java 바이트 코드에 대한 역컴파일이 가능한 취약성을 지니게 된다. 물론 Decrypt 모듈의 전송에 있어 Decrypt 모듈 코드에 대한 네트워크 상에서의 캡쳐도 가능하다. 그러나, Decrypt 모듈이 Stand-Alone 으로 실행 가능한 형태가 아닌 Java 실행 환경의 클래스 로더에 동적으로 구성되어 실행되는 형태로 구성될 경우 캡쳐된 Decrypt 모듈에 대한 악의적인 이용 가능성은 매우 줄어든다.
[그림 2]는 Java 실행 환경에서 암호화된 Java 바이트 코드의 처리 과정 및 클래스 로더와 Decrypt 모듈 사이의 관계를 보여준다. 기존의 Java 실행 환경과는 다르게 클래스 로더는 Java 클래스 파일(xxx.class)이 아닌 암호화된 코드(xxx.crypt)를 입력 받는다. 입력된 암호화 코드는 클래스 로더에 동적으로 구성되어 있는 Decrypt 모듈을 통하여 Java 클래스 파일의 형태로 변환된다. 이렇게 변환된 클래스 파일은 기존 Java 실행 환경과 마찬가지로 Verifier를 거쳐 JVM에 의하여 Native 인스트럭션으로 번역되어 실행된다.

설계 원칙 2를 만족 시키기 위하여 클래스 로더의 입력이 Java 클래스 파일이 아닌 암호화된 코드(xxx.crypt)로 이루어지도록 구성하였다. 암호화된 코드의 해독이 클래스 로더 외부에서 실행되어 클래스 파일이 클래스 로더의 입력으로 이루어질 경우, 클래스 파일은 Java 실행 환경 외부 어딘가에 존재하여야 한다. 이는 곧 암호가 해제된 클래스 파일이 이용자의 기기에 저장될 수 있음을 의미한다. 이러한 가능성은 Reverse Engineering에 대한 취약함을 드러낼 수 있다. 그러므로, 암호가 해제된 Java 바이트 코드는 외부 저

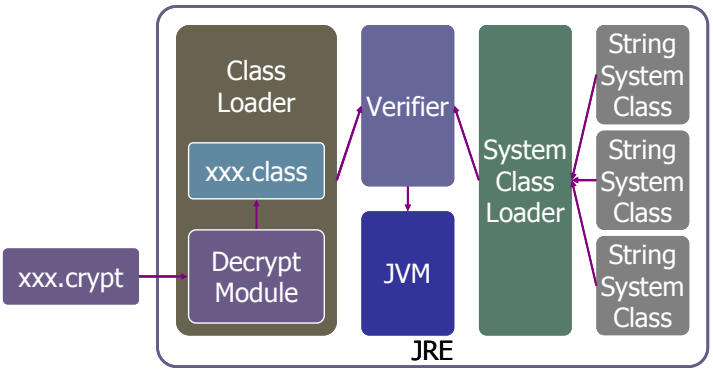

그림 2 역컴파일 방지 Java 실행 환경

장 매체가 아닌 시스템의 메모리에 적재되어야 하며, 이는 Java 실행 환경의 컨텍스트 상에서만 존재하여야 Java 바이트 코드를 이용자의 접근으로부터 차단할 수 있다. 이러한 방식으로 처리하기 위해 Decrypt 모듈은 반드시 클래스 로더 내부에서 호출되고 동작해야 한다.

Decrypt 모듈에 대한 클래스 로더 내부에서의 호출 및 실행을 보장하기 위하여 Decrypt 모듈을 적재하는데 있어 동적 인스트루멘테이션 기법을 적용한다. 동적 인스트루멘테이션 기법은 실행 중인 임의의 프로그램에 대하여 재컴파일, 재링크, 재실행의 단계를 생략하여, 내부에 새로운 기능을 Native 인스트럭션으로 삽입하여 추가, 수정하거나 특정 부분을 삭제할 수 있는 기법이다[11]. 이를 통하여 각각 암호화된 프로그램에 대하여 이용자의 기기로 1:1로 다운로딩되는 Decrypt 모듈을 동적으로 선택하여 암호 해제를 수행할 수 있게 된다.

이렇게 구성된 Java 실행 환경을 통하여 기존 암호화를 통한 Reverse Engineering 방지 기법에 존재하던 암호 키의 노출 및 암호 해제가 이루어진 바이트 코드의 노출이라는 단점을 보완할 수 있으며, 보다 신뢰성 있는 모바일 에이전트 환경을 구축할 수 있을 것이다.

4. 결론

현재의 네트워크화 된 환경에서 Reverse Engineering 방지의 중요성은 높아지고 있다. 본 논문에서는 이러한 환경에서 Reverse Engineering 방지에 대한 기존의 연구에 대하여 검토하고, Java 바이트 코드의 암호화를 통한 방어 기법으로서 동적 인스트루멘테이션을 이용한 재구성이 가능한 새로운 Java 실행 환경을 제안하였다. 이를 통하여 보다 신뢰성 있는 모바일 에이전트 환경을 구축할 수 있을 것이다.

참고 문헌

1. Walter Binder , Volker Roth, Secure mobile agent systems using Java: where are we heading?, Proceedings of the 2002 ACM symposium on Applied computing, March 11-14, 2002, Madrid, Spain
2. G.Naumovich, N.Memon, Preventing Piracy, Reverse Engineering and Tampering, IEEE Press, 2003
3. Jad - the fast JAva Decompiler, http://www.kpdus-.com/jad.html
4. Mocha, the Java Decompiler, http://www.brouha-ha.com/~eric/software/mocha/
5. Neeran M. Karnik , Anand R. Tripathi, Design Issues in Mobile-Agent Programming Systems, IEEE Concurr-ency, v.6 n.3, p.52-61, July 1998
6. Garfinkel, M. Rosenblum, and D. Boneh. Flexible OS Support and Applications for Trusted Computing. In Proceedings of the 9th Workshop on Hot Topics in Operating Systems, Kauai, Hawaii, May 2003
7. Sape Mullender, editor, Distributed Systems, Addi-son-Wesley, 2nd edition, 1993
8. Low, D. Protecting Java Code Via Code Obfuscation. ACM Crossroads, 4, 3 (Spring 1998).
9. D. Lie et al., Architectural support for Copy and Tamper Resistant Software, Proc. 9th Int' l Conf. Ar-chitectural Support for Programming Languages and Operating Systems, ACM Press, 2000
10. Trusted Computing Group, https://www.trustedc-omputinggroup.org
11. J. K. Hollingsworth, B. P.Miller, J. Cargille, Dynamic Program Instrumentation for Scalable Performance Tools, In Proceedings of the Scalable High Performance Computing Conference, 1994