유해 애플릿 공격에 대한 애플릿 보안 기술

박상길*, 노봉남*
*전남대학교 전산학과
e-mail:sgpark@athena.chonnam.ac.kr

Applet Security Technique against Hostile Applet's Attack

Sang-Kil Park*, Bong-Nam Noh*
*Dept of Computer Science, Chon-Nam National University

요 약

웹상의 자바 애플릿은 클라이언트의 웹 브라우저에 다운로드 되어서 브라우저 내부에 있는 자바가상기계(JVM : Java Virtual Machine)내에서 실행된다. 각 자바가상기계에는 실행 전에 바이트 코드 검증기와 바이트 코드 인터프리터를 통하여 오류문법을 점검한다. 애플릿을 이용한 잠재적인 공격형태는 시스템 수정, 개인정보의 침해, 서비스 거부공격, 강한 거부감을 느끼게 하는 공격이 있다. 이러한 유해한 애플릿의 공격에 대응하기 위한 방법으로 코드분석, 행위분석, 위치정보등을 이용한 보안기법이 제시되었지만 효율적인 대응을 하지 못하고 있다. 이 논문에서는 자바의 특성을 이용하여 자바클래스 내부의 바이트 코드 수정을 통한 애플릿 보안기술에 대해 기술한다. 유해한 행동이 예상되는 애플릿의 클래스에 대하여 바이트 코드 수정을 통하여 안전한 클래스로 대체함으로서 유해 애플릿 공격으로부터 시스템을 보호한다. 이를 수행하기 위해 프록시 서버를 두어서 웹브라우저의 요구를 수용하고, 이를 웹 서버에게 Safe클래스로 수정하여 요구하며, 그에 대한 응답도 처리한 후 애플릿에게 보여준다. 이는 런타임때 수행되며 웹브라우저, 서버, 클라이언트의 수정없이 프록시 서버의 개입으로 이루어진다.

1. 서론

자바(JAVA)프로그램은 시스템 개발과 웹 페이지의 능동적 요소등의 목적으로 개발된다. 악의적 코드의 실행을 막기 위해 애플릿과 자바 프로그램을 실행하기 전에 바이트 코드 검증기로 확인하고, 코드의 속성 등도 확인한다. 그러나, 이러한 방법도 런타임때 바람직하지 못한 서비스 거부 공격들을 막지는 못한다. 따라서 애플릿 속성에 영향을 미치는 메소드등을 사용자의 요구에 맞게 생성할 수 있는 방법이 필요하며, 이는 런타임 환경에서 바이트 코드 수정(bytecode modification)을 이용하여 수정된다. HTTP 프록시 서버를 이용하여 브라우저가 클래스를 수신하기 전에 프록시 서버가 클래스를 수정한다. 바이트 코드 수정시 추가되는 명령들은 자원의 사용을 점검하고 제어하며, 애플릿의 기능을 제한하며, 객체에 대한 접근제어를 제공한다. 위와 같이 점검하는 방법은 클래스 수준 수정과 메소드 수준 수정으로 구분된다. 바이트 코드 수준 수정은 자바로만 구현된 Muffin 프록시 서버를 사용하고, 바이트 코드의 제어를 위해 *JavaClass*를 설치한 뒤 Safe 클래스를 유형별로 추가한다.

2. 자바 애플릿 보안

자바 바이트 코드수정을 기술하기 전에 시스템에 해를 끼치며 바람직하지 못한 애플릿을 예로 들어보고, 그 문제점을 해결하는 방법에 대해서 살펴본다.

2.1 서비스 거부 공격

자바가상기계(JVM : Java Virtual Machine)는 서비스 거부 공격에 대한 보호기능은 거의 제공하지 못한다. 애플릿이 CPU 처리시간을 독점하여 시스템을 불안정하게 하거나, 계속된 메모리의 할당으로 시스

템을 정지시킬 수 있고, 쓰레드와 시스템 프로세스를 기아상태에 빠지게 할 수 있다. 사용자가 화면을 제어할 수 없을 정도로 수많은 윈도우를 만들어서 시스템을 정지시킨다. 화면에 열 수 있는 윈도우 수는 한계가 있고 임계치를 넘게되면 시스템은 폭주로 인해 멈추게 된다. 이는 실행환경에서 부적절한 시스템 자원의 사용으로부터 자바의 안전성이 위협될 가능성이 있으므로 자원의 사용과 제어에 대한 감시 메커니즘이 필요하다.

2.2 기밀정보의 유출

넷스케이프, 익스플로러, 핫자바와 같은 웹 브라우저들은 애플릿이 로드되어지는 곳의 유형에 따라 애플릿에게 서로 다른 권한을 부여한다. 그러나, 애플릿은 사용자의 기밀정보를 URL 리다이렉트등의 위장 채널을 사용하여 전송할 수 있으며, 애플릿은 브라우저에게 웹의 특정 페이지를 로드하라고 명령할 수 있다. 또한 이용 가능한 또다른 채널은 애플릿이 인터넷에 연결되어있는 서버에서 전자우편을 보내는 것이다. 웹서버가 SMTP 전자우편 데몬을 운영중이라면, 애플릿은 웹 서버의 25번 포트에 접속한 후에 *sendmail*과 메시지를 주고받음이 가능하다. 이러한 허점을 이용해 악의적인 애플릿은 전자우편을 위조할 수 있다. 전자우편의 위조를 막는 방법은 25번 포트에 접속을 허락하지 않는 것이다.

인터넷을 통하여 개인적인 정보의 접근이 금지되어 있는 A 애플릿은, 변수 등을 공유하는 B 애플릿 같은 다른 애플릿에게 정보를 전달할 수 있다. 이러한 저장매체를 이용한 애플릿 상호간의 통신은 감사장치를 통한 애플릿의 행위를 모니터링 함으로써 감지할 수 있다.

2.3 스푸핑(spoofing) 공격

위장 공격에서 공격자는 사용자가 보안과 관련된 부적절한 결정을 하도록 속이기 위해서 잘못된 정보를 만든다. 애플릿은 마우스가 그림이나, 링크 위를 지날 때 접근 가능한 URL을 브라우저 하단부의 상태바에 보여주게 된다. 사용자가 단순히 애플릿이 잘못된 URL을 상태바에 보여주는 것만 믿으면 공격자의 의도에 속게되며 이를 통해 애플릿은 사용자로하여금 다소 위험한 사이트로 접속하게 만든다. 위장공격의 경우는 상태바에 표현되는 URL에 대한 제어를 통해 해결 가능하다.

2.4 괴롭힘(Annoyance) 공격

자바의 장점 중 하나인 배경음악을 연주하는 기능을 파헤친 공격으로 시끄러운 소리를 그치지 않고 반복하여 들리게 하여 사용자를 짜증나게 할 수 있다. 이를 해결하기 위해서는 소리를 연주하는 쓰레드를 죽이거나, 오디오를 실행불능상태로 하거나, 브라우저를 종료하는 것으로만 가능하다. 괴롭힘 공격 중 가능한 방법은 브라우저가 방문한 사이트를 계속 반복해서 방문하도록 하는 것과, 그때마다 새로운 윈도우를 생성하는 것이다. 사운드에 대한 괴롭힘을 해결하는 방법은 사운드를 끔으로서 해결이 가능하다. 그 외의 해결책으로는 자바 실행환경에서 사운드 객체에 대한 모니터링과 제어가 꼭 필요하다.

3. 자바 바이트코드 수정

앞에서 살펴보았던 공격의 해결책으로 바이트코드 수정 메커니즘을 살펴본다. 기본적 개념은 애플릿 코드에 Safe코드를 삽입함으로써 제한을 가하는 것이다. Safe코드는 애플릿의 기능을 제한할 뿐만 아니라, 자원의 사용에 대해 모니터링과 제어를 한다. Safe 메커니즘은 실행환경에서 테스트하였을 경우, 클래스 또는 메소드와 같은 실행가능한 요소로서 대치된다. 이러한 Safe 메커니즘은 반드시 애플릿이 실행되기 전에 수행되어 바이트코드를 변경해야 한다. 애플릿은 웹 서버와 클라이언트의 브라우저 사이에 위치하는 HTTP 프록시(Muffin) 서버에 의해 변경된다. 이러한 방법은 웹서버와 JVM, 웹 브라우저에 대해 어떠한 변화도 없이 구현되는 장점이 있다.

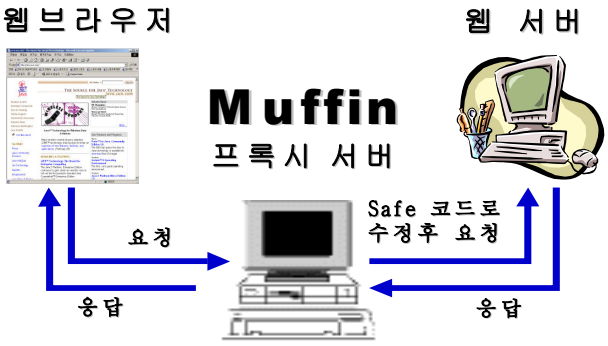

그림 1 Muffin 프록시 서버의 HTTP 처리

애플릿과 브라우저는 다운로드된 애플릿코드의 수정을 알지 못하고, 브라우저의 요구는 프록시 서버를 통하여 Safe코드로 변경된 후 웹 서버에게 요구하게 된다. 이러한 문제는 Safe 코드가 저장하고 있는 곳에서의 요구가 프록시 리다이렉트에 의해 이루어진다.

3.1 클래스 수준 수정(Class-level Modification)

*Window*와 같은 클래스의 경우 탑레벨 클래스인 *Frame*클래스는 좀 더 제한적인 *Safe$Frame*으로 바뀌어진다. 이는 추가적인 보안과 온전함을 확인하는 루틴을 포함하고 있다.

Safe$Frame 클래스의 메소드 생성자는 화면에 몇 개까지의 윈도우가 열리도록 제한할 수가 있다. 이 메소드의 사용으로 임계치를 초과하여 윈도우를 열려고 할 때까지는 새로운 윈도우의 생성은 허락된다. *Safe$Frame*클래스는 *Frame*클래스의 서브타입이며, *Frame*클래스가 요구되는 모든 곳에 대신하여 존재하게 된다. 결국 애플릿은 임계치 이상으로 윈도우를 생성하기 전까지는 그 변화를 알 수 없다.

위와 같은 클래스 수준수정은 단순히 *Frame* 클래스에 대한 참조를 *Safe$Frame*으로 바꿈으로서 이루어진다. 자바는 모든 문자열과 클래스, 필드, 메소드들에 대한 참조는 클래스 파일의 상수풀(constant pool)을 통해 결정된다. 상수풀에는 클래스 이름이 어느 곳에 저장되어 있는지를 나타낸다. 그림 2와 같이 상수 풀을 표현하는 요소로는 CONSTANT_Class, CONSTANT_Utf8이 있으며, CONSTANT_Class는 클래스의 전체이름을 표현하는 문자열 UTF-8의 요소인 CONSTANT_Utf8을 참조한다.

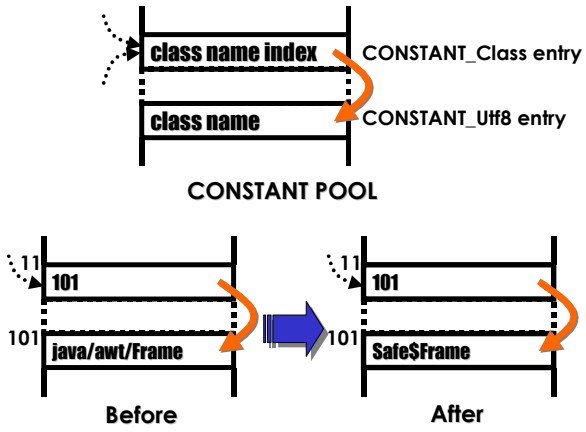

그림 2 클래스 수준의 클래스 수정

클래스 이름의 CONSTANT_Utf8의 요소 *java/awt/Frame*를 새로운 클래스인 *Safe$Frame*으로 바꾼다면, 그림2와 같이 CONSTANT_ Class 요소는 *Safe$Frame*클래스로 변경된다. 클래스 수준 수정은 상속성을 이용하여 단순하게 상수풀의 해당요소(entry)의 수정만을 요구한다. 그러나 클래스의 상속성을 이용할 때 final 클래스와 인터페이스의 경우에는 자바언어에서 상속되지 않으므로, 클래스 수준 서정을 할 수 없다.

3.2 메소드 수준 수정(Method-Level Modification)

클래스 수준의 수정을 할 수 없는 final키워드나 interface를 통해 접근 가능한 메소드의 경우 메소드 수준 수정이 사용되며, 좀더 복잡한 메소드 참조와 메소드 유도 명령이 요구된다.

바이트코드 수준에서 기술자 필드는 표 1과 같이클래스나 인스턴스 변수의 형식을 표현한다. 예를 들어 int 인스턴스의 경우 간단히 I로 표현하며 클래스의 instance의 경우는 L<classname>;으로 표현한다.

표 1 필드 기술자의 의미

Descriptor	Type
C	Character
I	Integer
Z	Boolean
L<classname>;	An instance of the class

메소드 기술자는 메소드가 갖는 매개변수와 반환되는 값을 표현한다. 매개변수 기술자는 0 또는 그 이상의 필드 타입들을 갖는다. V 문자는 메소드의 리턴 값이 없을 때(void)를 나타낸다. 메소드 void foo(Thread t, int i)는 (Ljava/lang/Thread;I)V 로 표현된다. 메소드 유도 명령이 어떻게 변경되는지 보기전에, method가 클래스파일로 어떻게 컴파일되어지는지 그림 3을 통해서 알아본다.

예제 메소드

```
void ex(Thread t, int i)
t.setPriority(i);
```

컴파일된 메소드

```
method public example1(Ljava/lang/Thread;I)V
push Ljava/lang/Thread;I)V
push I
invokevirtual Thread.setPriority(I)V
```

그림 3 예제 메소드와 컴파일된 메소드

Thread.setPriority(I)V를 조금 더 제한적인 method인 Safe$Thread.setPriority(Ljava/lang/Thread;I)V로 교체한다. 교체된 Safe$Thread 클래스에 정의되는 setPriority 메소드는 applet에게 상위한계치보다 큰 우선순위를 부여할 수 없도록 한다. 새로운 Safe코드 메소드는 Thread 클래스의 메소드의 인스턴스를 발생시킨다. 예를 들면, t.setPriority(5)는 Safe$Thread.setPritority(t,5)로 된다. 새로운 메소드는 priority를 integer형 매개변수로 취하여 상위 한계치와 비교한다. 매개변수의 값이 상위한계치보다 크면 매개변수의 값이 상위한계치가 된다.

3.2.1 메소드 참조 수정

클래스 또는 클래스 인스턴스의 메소드는 상수풀 요소에 CONSTANT_Methodref로서 표현되어진다. 그림 4와 같이 CONSTANT_Methodref 엔트리는 메소드가 클래스의 멤버인지를 표현하는 CONSTANT_Class, 메소드의 기술자 표현하는 CONSTANT_NameAndType을 참조한다.

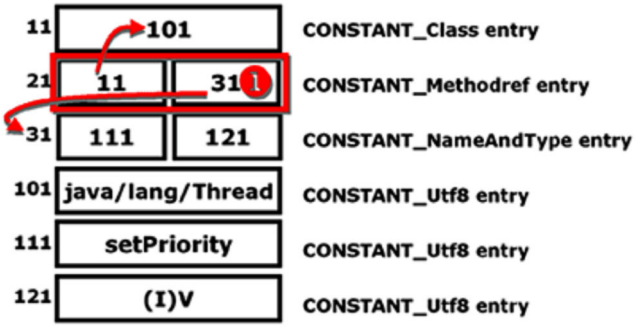

그림 4 Thread.setPriority(I)V에 대한 참조

그림 5와 같이 새로운 Safe$Thread 클래스를 나타내기 위해 클래스 이름을 나타내는 CONSTANT_Utf8 요소를 (201번지) 추가하고, 새로운 CONSTANT_Class(21번지) 요소는 CONSTANT_Utf8(202번지) 요소를 참조하도록 추가하였다. 그러면 CONSTANT_Methodref 요소는 새로운 CONSTANT_Utf8 요소(201번지)를 참조하도록 수정된다.

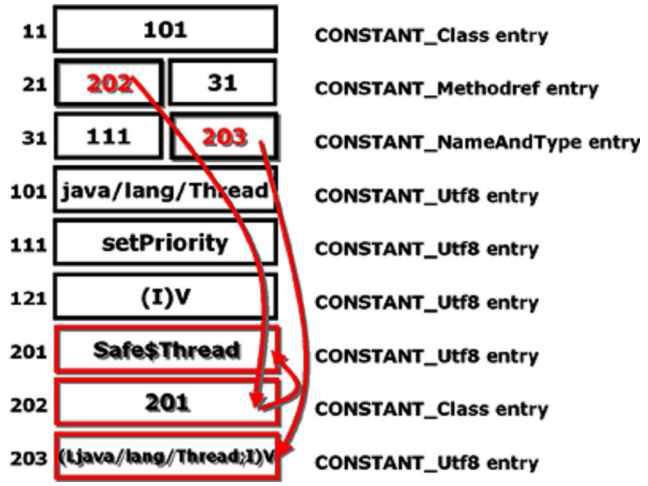

그림 5 Safe$Thread.setPriority(Ljava/lang/Thread;I)V로의 참조

결과적으로 java/lang/Thread대신에 Safe$Thread를 참조하도록 수정된다. 메소드 기술자가 바뀌었으므로, 새로운 메소드 기술자(Ljava/lang/Thread;I)V에 대한 참조를 위해 CONSTANT_Utf8요소를 추가한다. 그러면 CONSTANT_NameAndType요소는 새로운 메소드 기술자를 위한 CONSTANT_Utf8요소로의 참조로 수정된다. 위와 같은 단계 후에, CONSTANT_Methodref 요소는 위의 그림 5와 같이 Safe$Thread.setPriority(Ljava/lang/Thread;I)V라는 새로운 메소드를 표현한다.

3.2.2 메소드 호출 명령 수정(Method Invoking Instruction Modification)

메소드 호출을 구현하는 다양한 자바가상기계 instrucion중에서 2가지를 살펴본다. ***invokevirtual*** 은 클래스 인스턴스의 메소드 호출을하며, ***invokestatic*** 은 static 클래스의 메소드 호출을 한다. 클래스 메소드 호출은 단순히 피연산자 스택으로 푸쉬되는 매개변수만이 필요하다.

Thread.setPriority(Ljava/lang/thread;I)V는 ***invokevirtual***에 의해 피연산자 스택에 푸쉬된다. 메소드 수정을 수행하면 ***invokevirtual***에 의해 피연산자 스택에 푸쉬되던 작업이 ***invokestatic***에 의해 수행되도록 바이트코드가 바뀌어지며 그 결과 메소드가 수정되어 Safe$Thread.SetPriority(Ljava/lang/Thread;I)V로 된다. 클래스 수준 수정은 단순히 상수풀의 수정으로 가능하며, 메소드 수준 수정은 바이트코드 명령과 상수풀 요소의 수정으로 가능하다. 클래스 수준 수정은 ***final*** 클래스 또는 인퍼테이스(**interface**)에는 적용될 수 없으므로 그에 대한 방법으로 메소드 수준의 수정이 이루어 져야 한다.

4. 바이트코드 수정을 통한 악의적 공격에 대한 대응

앞에서 살펴보았던 악의적인 애플릿의 공격에 대한 바이트코드 수정기법을 이용한 대응책을 살펴본다.

4.1 Window 클래스를 이용한 서비스 거부 공격

윈도우 시스템이 제어할 수 있는 윈도우 보다 더 많은 윈도우를 생성하는 애플릿은 시스템을 정지시킬 수 있다. 이러한 자원 소모형 공격에 대한 대응책으로 윈도우 생성에 대한 감시를 수행하는 Safe 메커니즘이 있어야 한다.

자바 라이브러리 클래스인 *Frame*은 *Window*클래스의 최상위(Top-level)클래스로서 윈도우를 제어할 수 있다. 이러한 공격에 대한 주요한 해법은 정해진 수 이상으로 생성자(Constructor)를 호출할 수 없도록 제한함으로서 애플릿 공격을 허락하지 않는다.

*Frame*이 *final* 클래스가 아니므로 *Safe$Frame* 클래스가 생성될 수 있다. 이는 모든 *Window* 클래스의 인스턴스 생성을 감시하고 제어하게 된다. 주어진 윈도우의 수를 넘어서 윈도우를 생성할 수 없도록 제어한다. *Frame*으로의 참조를 *Safe$Frame*으로 변환하는 클래스 수준 수정을 사용한다. 이 방법을 또한 윈도우의 크기와 윈도우의 위치에 대해서도 제어가 가능하다.

4.2 전자메일을 위조한 공격에 대한 대응

인터넷에 연결되어 있는 웹 서버에 SMTP sendmail 데몬이 설치되어있으면, 유해한 애플릿은 사용자의 중요한 정보를 유출할 수 있다.

자바 라이브러리 클래스인 *Socket*은 네트웍을 통한 프로세스간의 통신을 위한 소켓을 구현한다. 생성자 메소드는 소켓을 생성하고, 확인된 호스트와 포트에 연결한다. 소켓을 통한 공격을 제한하기 위해서는 생성자 메소드를 제한하여야 한다. 자바가상기계에 구현되는 생성자 메소드 호출명령에 대해 살펴보자. 자바가상기계 클래스 인스턴스들은 JVM의 ***new*** 명령에 의해 생성된다. 클래스 인스턴스가 생성될 때 인스턴스 변수들은 그들의 기본값으로 초기화된다. 새로운 클래스 *instanc<init>*의 인스턴스 초기화 메소드가 호출된다.

다음 예제를 통하여 살펴보자.

Example

```
Socket create() {
        return new Socket(host_name, port_number);
}
```

컴파일 결과

Method java.net.Socket create()

0	new #1	Class java.net.Socket
3	dup	
4	getfield	Field this.host_name java.lang.String
7	getfield	Field this.port_number I
10	invokespecial #4	Method java.net.socket.<init>(Ljava/lang/String;I)V
13	areturn	

그림 6 인스턴스 초기화 메소드의 예

invokespecial은 인스턴스 초기화 메소드 호출을 위한 JVM의 명령어이다. 이는 superclass, private, 인스턴스 초기화 메소드들과 같은 특별한 제어가 필요하는 인스턴스 메소드들을 발생한다.

Socket이 웹 브라우저에서 final클래스이므로, 메소드 수준 수정을 통해 메소드 생성자를 수정해야 한다. Static Safe메소드인 *Safe$Socket.init*은 모든 소켓 연결을 감시하고 제어한다. *Safe$Socket.init*은 25번 포트를 제외한 모든 요구에 대해 소켓 연결하며, socket 객체를 반환한다. *Safe$Socket.init*은 생성자 Socket이 갖는 것과 동일한 매개변수 들을 취하지만, 새로운 소켓 객체가 반환되면 다른 return type을 반환한다.

그러므로, *Socket.<init>*(Ljava/lang/String;I)V에 대한 참조는 *Safe$Socket.<init>*(Ljava/lang/String;I)Ljava/net/Socket;으로 바뀌어야 한다.

*Safe$Socket.init*이 static 메소드 이므로 ***invokespecial***을 ***invokestatic***으로 바꾸어야 한다. 또한 ***new*** 메소드는 socket객체를 반환하기 때문에 ***new*** 에 의해 스택으로부터 생성되는 소켓 객체를 제거해야 한다. 수정된 코드는 다음과 같다.

Method java.net.Socket create()

0	new #1	Class java.net.Socket
3	**pop**	
4	getfield	Field this.host_name java.lang.String
7	getfield	Field this.port_number I
10	**invokestatic** #4	**Method Safe$Socket.<init> (Ljava/lang/String;I)Ljava/lang/Socket;**
13	areturn	

그림 7 수정된 Safe$Socket클래스

4.3 URL 위장(spoofing) 공격

애플릿은 상태바에 위장 URL을 표시하는 것을 이용하여 사용자를 속일 수 있다. 이러한 위장 공격은 URL에 보이는 것과 웹에 실제 올리어지는 URL의 일관성을 확인함으로서 막을 수 있다.

자바 라이브러인 인터페이스인 ***AppletContext***는 애플릿이 웹 브라우저 또는 애플릿뷰어의 context와 상호교환 할 수 있도록 허락하는 메소드들을 정의한다. ***showDocument*** 메소드는 브라우저/애플릿 뷰어가 URL 매개변수에 의해 지시되는 웹 페이지를 나타내도록 요구한다.

showStatus는 웹브라우저/애플릿 뷰어의 상태바에 텍스트를 나타내는 메소드이다. *Safe$AppletContext.showStatus*는 *Safe$AppletContext.showDocument*가 나중에 참조할 수 있도록 현재 현재텍스트를 상태바에 보이고 저장한다. *Safe$AppletContext.showDocument*가 유도되면 URL 매개변수가 현재 보여지는 상태바의 텍스트와 일치하는가 확인하다. 일치하지 않는다면 요구를 통과시키는 대신에 상태바에 URL 매개변수를 보인다. 후자의 경우 사용자는 비일관성을 알게되고 그에 적절한 행동을 취하게 될 것이다. 일반적으로 사용자는 한번 이상의 마우스 클릭으로 웹페이지를 로드하게 된다. 이는 사용자가 상태바에 표시되는 URL과 함께 새로운 웹 페이지를 가져오는 것을 보장한다.

*AppletContext*인터페이스는 상속이 불가능하므로, 2개의 메소드는 메소드 수준 수정을 통해 수정된다.

invokeinterface는 인터페이스 메소드를 호출하는 명령이다. 인터페이스 메소드가 static method로 교체

되었으므로, *invokeinterface*는 *invokestatic*으로 바꿔어야 한다. 또한 *invokestatic*은 *invokeinterface*의 2개의 피연산자를 갖지 않으므로 2개의 피연산자는 *nop* 명령을 할당한다.

4.4 괴롭힘 공격에 대한 대응

애플릿은 끊임없는 사운드를 통해 사용자를 괴롭힐 수 있다. 이러한 괴롭힘 공격을 막는 것은 사용자가 사운드를 off함으로써 가능하다. 그러나 이렇게 하면 시스템 전체의 사운드를 off하는 것이 된다.

자바 라이브러리 인터페이스인 *AudioClip*은 사운드를 연주하는 핵심적인 메소드를 표현한다. *Appletcontext.getAudioClip()*과 *Applet.getAudioClip()*은 모두 이 인터페이스를 구현한 객체를 반환한다. 객체의 *loop* 메소드는 루프 내에서 사운드 연주를 시작하고, *stop* 메소드는 audio clip을 연주하는 것을 멈춘다. 이 유형의 공격은 사운드 loop를 형성하여 결코 stop하지 않도록 구현한다. 애플릿이 이 객체의 loop 메소드를 호출할 때마다 Safe 메커니즘은 소리를 off할 수 있는 *stop* 메소드가 있는 창을 띄운다.

AudioClip 인터페이스가 상속이 불가능하므로, Safe 메커니즘은 메소드 수준 수정을 사용한다. *AudioClip.loop()V*에 대한 참조는 *Safe$AudioClip.loop()V*로 바뀌며 *AudioClip.stop()V*에 대한 참조는 *Safe$AudioClip.stop()V*로 각각 바뀐다.

그리고 *invokeinterface*는 *invokestatic*으로 수정되며 2개의 추가적인 매개변수들은 *nop* 명령으로 수정된다.

5. 결론

본 논문에서는 애플릿의 유해한 행동을 소스코드로 역컴파일(Decompile)하여 유형을 판단하는 차원에서 탐지하는 방법을 사용하는 것이 아니라, 'Java Virtual Machine Specification'에서 제시하는 클래스 파일의 형식을 이용하여 바이트코드 상태에서 클래스수준 수정을 수행하여 유해한 행동을 사전에 제한하는 바이트코드 수준 수정에 대하여 기술하였다. 바이트코드 수준 수정은 클래스 수준 수정과 메소드 수준의 수정이 있다. 상속가능한 클래스의 경우에는 클래스 수준을 사용하며 final 클래스와 interface들에 대하여서는 적용이 불가능하여 메소드 수준 수정한다. 메소드 수준 수정은 바이트코드 명령과 상수풀을 수정하여 이루어진다. 이 방법을 이용하면 서버, 클라이언트, 브라우저에 대해 어떠한 별도의 작업도 필요하지 않고, 프록시 서버에서 사전에 정의한 유해한 행동을 제어하는 클래스에 의해 수정된 후 브라우저에 보여진다. 그러나, 프록시 서버를 사용한 면과 자바로 구현된 서버인 관계로 속도측면에서 상당히 부하가 작용된다. 그리고, 온라인으로 패턴이 추가되지 않아서 직접 Safe 클래스에 추가하여야 한다는 단점이 있으므로 이에 대한 보완작업이 필요하다.

6. 참고 문헌

[1] Dirk Balfanz and Edward W.Felten, A Java Filter, *Technical Report 97-567*, Department of Computer Science, Princeton University, 1997.

[2] Drew Dean, Edward W.Felten and Dan S.Wallach, Java Security : From Hotjava to Netscape and beyond, In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[3] Edward W.Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach, Web spoofing : An Internet Con Game, *Technical Report 540-06*, Department of Computer Science, Princeton University, Feb 1997.

[4] Tim Lindholm and Frank Yellin, The Java Virtual Machine Specification 2nd Edition, http://java.sun.com/docs/books/vmspec/index.html

[5] Gary McGraw and Edward W. Felten, Java Security : Hostile Applets, Holes, and Antidotes, John Willey & Sons, 1997.

[6] David M, Martin Jr, Sivaramakrishnan Rajagopalan, and Aviel D. Rubin. Blocking Java Applets at the Firewall., In P*roceedings of the 1997 Internet Society Symposium on Network and Distributed System Security*, Feb 1997.

[7] G.C Necula and Peter Lee. Safe kernel extensions with run-time checking. In *Proceedings of the 2nd Symposium on Operating Design and Implementation*, Oct 1996.

[8] M. Dahm, Byte Code Engineering with the JavaClass API, Technical Report B-17-08, Freie Universitat Berlin, 1998.

[9] Gary McGraw, Edward W. Felten, SecuringJava : Getting Down to Business with Mobile Code, http://www.securingjava.com, 1999.

[10] Inshik Shin, John C. Mitchell, Java Bytecode Modification and Applet Security, http://theory.stanford.edu/~vganesh/project.html