자바 바이트 코드를 이용한 인터넷 통신의 애플릿 제어

정회원 김 문 환*, 나 상 동

A Study on Applet Control on the Internet Communication using Java Bytecode

Moon-Hwan Kim*, Snag-Dong Ra *Regular Members*

요 약

웹 브라우저에서 자바 애플릿 파일은 시스템의 가상머신에 의해 클라이언트 브라우저의 가상 머시인을 실행한다. 자바애플릿을 실행하기 전에 자바 가상머신은 bytecode 수정자를 이용하여 bytecode 프로그램을 검색하며 해석기를 이용하여 실시간 테스트를 수행한다. 그러나 이러한 테스트들은 서비스 거부공격, 이메일 위조 공격 URL 추적공격 또는 지속적인 사운드 공격과 같은 원하지 않은 실행시간 동작을 예방할 수 없다.

본 논문에서는 이러한 애플릿을 보호하기위해 자바바이트 코드 수정기술이 사용한다. 수정기술은 검사를 수행할 적절한 바이트코드를 삽입함으로서 애플릿 동작을 제안한다. 자바 바이트 수정은 두개의 형태로 분류되며 클래스 레벨 수정은 마지막 크레스가 아닌 서브크레스를 포함하기 때문에 메소드 레벨 수정은 마지막 클래스 아닌 서브크레스를 포함한다. 메소드 레벨 수정은 마지막 클래스 또는 인터페이스로부터 객체들을 제어할수 있다.

본 논문은 악성 애플릿들이 프록시 서버를 이용한 자바 바이트 코드 수정에 의해 제어되는 것을 나타냈으며 이러한 구현은 웹 서버, JVM, 웹 브라우저상에서 악성 애플릿들의 공격이 제어됨을 입증한다.

key Words : Java Applet, JVM(Java Virtual Machine), JDK(Java Development Kit), Bytecode Verification

ABSTRACT

Java applets are downloaded from web server through internet and executed in Java Virtual Machine of clients' browser. Before execution of java applets, JVM checks bytecode program with bytecode verifier and performs runtime tests with interpreter. However, these tests will not protect against undesirable runtime behavior of java applets, such as denial of service attack, email forging attack, URL spoofing attack, or annoying sound attack.

In order to protect malicious applets, a technique used in this paper is java bytecode modification. This technique is used to restrict applet behavior or insert code appropriate to profiling or other monitoring efforts. Java byte modification is divided into two general forms, class-level modification involving subclassing non-final classes and method-level modification used when control over objects from final classes or interface.

This paper showed that malicious applets are controlled by java bytecode modification using proxy server. This implementation does not require any changes in the web sever, JVM or web browser.

I. 서 론

인터넷 통신의 환경 속에서 서로 다른 플랫폼 간의 네트워크를 통하여 데이터베이스 응용, 멀티미디어 기능, 암호화 프로토콜 등의 기능을 제공한다. 인터넷 통신에서 애플릿(Applet)이라

* KTF 네트워크 교환운용팀, 조선대학교 컴퓨터공학부(sdna@mail.chosun.ac.kr)
논문번호 : 020519 - 1204, 접수번호: 2003년 12월 4일

는 실행코드를 서버로부터 다운로드받아 클라이언트에서 실행함으로써 서버와 클라이언트가 작업 부하를 나누는 분산처리 능력과 다양한 플랫폼 실행이 가능한 특징이 있다.

네트워크에서 서버로부터 다운로드받은 애플릿을 클라이언트에서 바로 실행할 경우 발생할 수 있는 여러 보안 문제에 노출되어 있기 때문에, JDK(Java Development Kit)를 개발할 때마다 보안 문제의 허점도 있다.

악의적인 코드의 실행을 막기 위해 자바 가상 머신(Java Virtual Machine)[1,2,3]에서는 애플릿과 자바 프로그램을 실행하기 전에 코드 속성을 검증하고, 추가적으로 실행 할 때도 검사한다.
그러나, 이러한 방법도 서비스 거부 공격, 지속적인 사운드 공격 등과 같이 원하지 않는 동작을 예방할 수는 없다.

본 논문은 인터넷 통신에서 자바 바이트 코드 수정(Bytecode Verification) 기술[4,6]을 이용하여 필요한 실행 환경에서 검사를 수행할 부가적인 바이트 코드 명령을 삽입함으로써 애플릿의 동작을 제한한다. 이러한 부가적인 명령을 통하여 사용량을 통제 및 제어하고, 애플릿을 기능적으로 제한하거나 접근할 수 없는 객체에 대한 제어를 제공한다. 클래스 계층 구조의 제한이 없는 메소드 레벨 수정을 이용하고, 메소드 레벨 수정은 바이트 코드 수정과 상수 풀의 수정을 하며, 자바 애플릿으로 동작하는 정보 연결에 제어되도록 한다. 또, 특정한 사이트들에 대한 접근을 막고, 특정한 자바 클래스들에 대한 요청을 수정하도록 애플릿 보안에 관하여 연구한다.

Ⅱ. 자바 애플릿 보안 처리

A. 자바 보안 모델과 애플릿 동작

그림 1에서와 같이 자바 프로그램을 수행하기 위해서는 먼저 자바 원시 프로그램을 자바 컴파일 한다. 자바 컴파일러는 자바 원시 프로그램을 자바 바이트 코드(Bytecode)로 플랫폼을 독립적인 중간 코드 형태의 프로그램으로 변환하고, 자바 인터프리터는 자바 바이트 코드로 수행하며, 한번 컴파일된 자바 프로그램은 인터프리터에 의해 반복적으로 수행된다.

그림 2에서는 자바 바이트 코드 "Write Once, Run Anywhere"[2,7]는 자바의 특징을 구현하는 핵심이며, 자바 컴파일러가 제공되는 모든 플랫폼에서 자바 원시 코드를 컴파일하여 플랫폼을 독립적인 바이트 코드로 만들 수 있다. 자바 바이트 코드는 어떤 플랫폼의 자바 가상 머신에서도 수행될 수 있기 때문에 동일한 바이트 코드는 Windows NT, Solaris, Macintosh에서 그대로 수행한다.

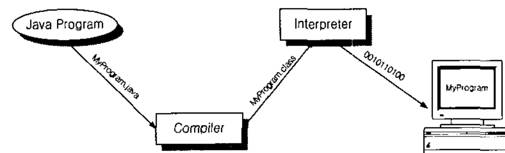


그림 1. 자바 프로그램의 수행과정
Fig. 1 Execution Process of Java Program

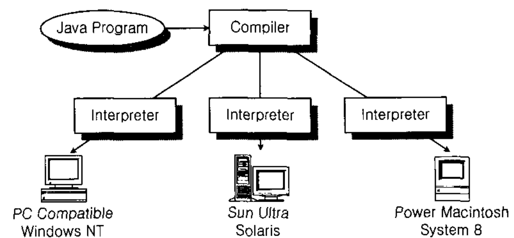


그림 2. "Write Once, Run Anywhere"를 보여주는 Java 수행 모델
Fig. 2 Java Execution Model Process of "Write Once, Run Anywhere"

JDK에서는 시스템 자원에 대한 접근 제한을 위해 JVM(Java Virtual Machine)내에 Sandbox라는 공간을 만들어 안전한 시큐리티를 제공한다[5,7].

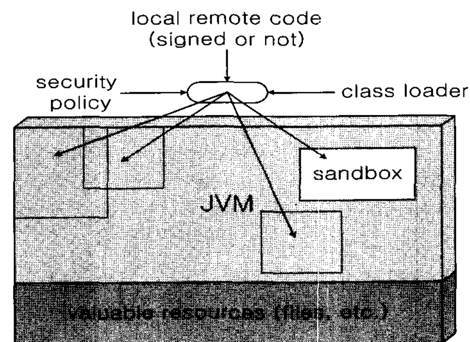


그림 3. JDK 1.2 보안 모델
Fig. 3 JDK 1.2 Security Model

JDK 1.2 보안 모델은 그림 3과 같이 현재 설정된 보호 정책(Security Policy)을 애플릿 코드

가 로드될 때 각 애플릿에 권한을 할당하고, 애플릿은 할당된 권한에 따라 사용자 자원에 접근이 가능하다[3,8]. 각 권한은 특정한 파일이나 디렉토리에 읽기나 쓰기 등 호스트나 포트에 허용된 접근을 지정한다.

내부적으로는 시스템 영역 내에 "Protected Domain"이라는 것을 두고 애플릿의 특성에 따라 다중 보호 정책을 설정할 수 있게 함으로써 유연성과 안전성을 강화[15,16]해서 하나의 에러가 전체적인 시스템 관리 부분에 치명적인 위험을 주는 것을 막을 수 있다.

B. 자바 애플릿 절차

자바 애플릿은 HTML 페이지에 포함되어 자바 호환(java-compatible) 웹 브라우저에 의해 실행될 수 있는 자바 프로그램이기 때문에 자바 호환 웹 브라우저가 자바 애플릿이 포함된 HTML 페이지를 보여줄 때, 웹 서버 쪽에 있는 자바 애플릿 코드를 다운로드 한 후 브라우저 내의 특정 영역에서 실행하게 된다. 이렇게 자바 애플릿을 HTML 페이지에 삽입하기 위해서는 <applet> 태그를 사용한다. 이러한 자바 애플릿이 악의적으로 사용될 경우[7,8] 사용자 시스템에 유해한 동작을 유발시킬 수 있다.

애플릿은 그림 4와 같이 클라이언트나 웹 브라우저 측에서 웹 서버측의 해당 페이지(URL)을 요구하게 되면 클라이언트 시스템으로 전송된다. 자바 애플릿은 클라이언트 시스템으로 전송된 후 클라이언트 측의 인터프리터에 의해 해석되어 특정한 기능을 수행한다.

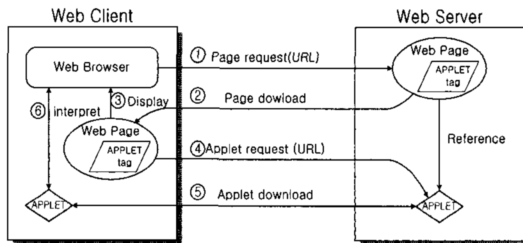


그림 4. 웹 상에서 애플릿 동작 절차
Fig. 4 Execution Process of Application The Web

C. 자바 애플릿 보안 과정

자바 애플릿을 다운로드받아 실행시키기 위해서는 자바 applet viewer나 자바를 지원하는 웹 브라우저를 사용하고, 자바 애플릿은 웹 브라우저에서 제공되는 sandbox에 의한 애플릿 실행 환경에서만 활동할 수 있다. 네트워크를 통해 전송된 애플릿은 신뢰성이 없는 것으로 간주하여 반드시 sandbox 내에 있는 보안에 따라 시스템 자원에 대한 접근을 제한한다.

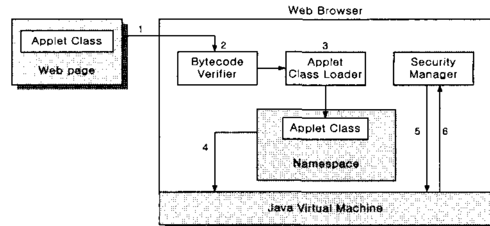
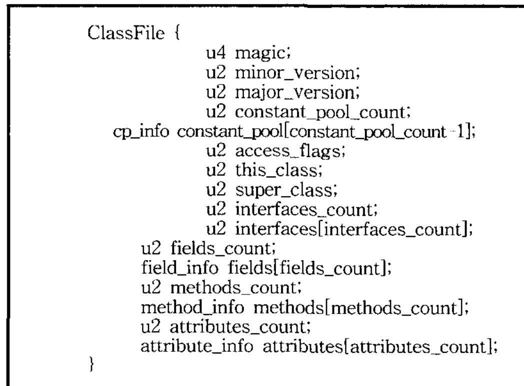


그림 5. 자바 애플릿의 보안 처리 과정
Fig.5 Security Management Process of Java Applet

그림 5에서와 같이, 먼저 웹 페이지에서 자바 코드를 가지고 오면, 바이트 코드 검사기를 사용하여 그것을 검사하고, 코드에 이름 공간을 할당하고, 애플릿을 실행한다. 위험한 메소드를 호출하면, 메소드가 실행되기 전에 보안 관리자를 참조한다. 보안 관리자는 호출하는 클래스의 근원지에 근거하여 검사하고, 애플릿의 일부 동작들을 제한하는 기능이 된다.

바이트 코드로 구성된 애플릿은 클래스 로더에 의해 실행되기 이전에 바이트 코드 검사기에 의해 먼저 검증을 받는다.

공격자가 사용자로 하여금 보안에서 URL은 상태 표시줄의 특정 위치에 나타내므로 애플릿이 위장된 URL을 표시되면, 사용자는 원하지 않는 결과를 얻을 수 있다. 악성 애플릿에서 끝나지 않는 매우 시끄러운 소리를 사용하여 정보를 괴롭힐 수 있다. 이러한 형태의 사운드 공격은 배경음악으로 사운드를 실행시킬 수 있는 자바의 특징을 사용한다.



```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

그림 6. ClassFile의 구조
Fig.6 Structure of Class File

그림 6의 ClassFile 구조에서 u2는 unsigned 형의 2byte 길이를 의미하고, u4는 unsigned 형의 4byte를 의미하며, 자바 클래스 파일은 Class 이름과 superClass의 이름, 구현된 interface의 이름, 정의된 항목들의 이름과 타입, class에 의해 제공되는 메소드를 위한 JVM 코드, 클래스에 의해 사용되는 상수 배열 값, JVM에 의해 사용되는 실행시간 정보 등을 포함한다.

표 1. 상수 풀 엔트리의 값
Table. 1 Value cf Constant Pool Entry

Constant Type	Value
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

표 1과 같이 자바의 모든 문자열과 클래스, 필드, 메소드들에 대한 참조는 클래스 파일의 상수풀(constant pool)을 통해 결정되고, 상수풀은 클래스 이름이 어느 곳에 저장되어 있는 지를 나타내며, 모든 상수 풀(Constant Pool)은 엔트리를 갖는 테이블이다.

```
cp_info {
        u1 tag;
        u1 info[];
}
```

constant_pool 테이블의 각각 항목은 cp_info 엔트리의 1바이트 크기의 tag로 시작하고, info 배열의 내용은 tag의 값에 따라 달라진다.

III. 자바 바이트 코드에 의한 수정

본 장에서는 자바 애플릿에 대한 보안 문제들을 해결하기 위한 기법을 제안한다. 수정 알고리즘은 실행 가능한 바이트 코드 명령을 애플릿에 삽입하는 바이트 코드 수정방법을 이용하여 유해한 애플릿의 동작을 제한하고, 안전 메카니즘은 클래스나 메소드와 같은 하나의 실행가능한 요소에 부가적인 실행 테스트를 수행하는 요소로 대체한다.

Window와 같은 클래스는 부가적으로 안전성을 체크하는 것 보다 클래스 Safe $ Window로 대체해서 이 안전 메카니즘을 애플릿이 실행되기 이전에 적용시킨다. 웹 서버나 자바 가상 머신, 그리고 웹 브라우저에 어떠한 변화도 요구되지 않게 애플릿들과 브라우저 애플릿 변화들이 알려지지 않았기 때문에, 안전하게 실행 가능한 요소를 위해 웹 서버에서 수정한다.

A. Class 레벨 수정

Window와 같은 클래스는 자원의 사용량과 기능을 제한하는 Window의 서브클래스 Safe$Window로 대체한다. Safe$Window의 생성자(constructor) 함수는 화면상에 열릴 수 있는 전체 윈도우의 수를 제한할 수 있기 때문에 그 함수는 윈도우가 전체 수 제한을 초과할 때까지 만들어지도록 허용한다. Safe$Window가 Window의 서브클래스이므로, Safe$Window는 Window가 기대되는 어느 곳이나 사용되기 때문에 애플릿 제한을 초과하는 윈도우를 생성하려고 시도하지 않는 한 변화하지 않으므로 자바 클래스 파일 내에서 수정될 수 있는 상수풀이다.

그림 7에서 하나의 클래스는 클래스의 이름을 나타내는 UTF-8 문자열을 위해 CONSTANT_Utf8 요소를 참조하는 CONSTANT_Class이며 상수 풀 영역에 의해 나타난다.

그림 8은 CONSTANT_Utf8 요소의 클래스 이름이고, Window를 새로운 클래스로 이름이 되면 Safe$Window로 제어되기 때문에 CONSTANT_Class 요소가 새로운 클래스 Safe$Window로 수정되어 나타난다.

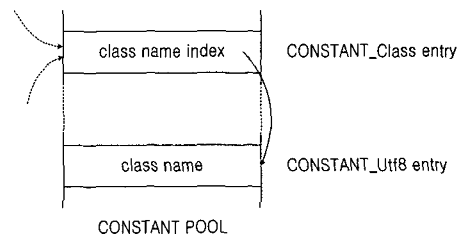


그림 7. 상수 풀 내에 두 개의 요소를 갖는 클래스
Fig. 7 A class Represented with to entries in the Constant Pool

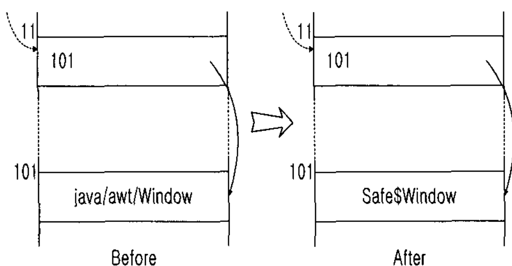


그림 8. 클래스 레벨의 수정
Fog. 8 Class-level Modification Substitute Class Reference

B. Method 레벨에 의한 수정

클래스 레벨 수정의 한계를 해결하기 위해서, 메소드 레벨 수정은 클래스 계층 구조를 사용하지 않고 메소드를 관련된 메소드로 대체한다. 메소드 레벨 수정은 메소드 수정과 메소드 호출 명령의 수정이 필요하다.

표 2에서와 같이 자바 클래스 파일 형식의 필드 정보는 클래스나 인스턴스 변수의 타입을 나타내므로 int 인스턴스 변수의 정보는 I로 표현한다.

표 2. 파일 형식
Table. 2. Field descriptor

Descriptor	Type	Interpreation
B	byte	signed byte
C	char	Unicode character
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L <classname>;	reference	an instance of class <classname>
S	short	signed short
Z	boolean	true or flase
[	reference	one array dimension

메소드 정보는 매개변수들과 반환하는 값으로 매개변수 정보는 0이나 더 많은 필드 타입이 되고, 반환값 정보는 필드 타입이나 V를 나타낸다. 문자 V는 메소드의 반환 값이 없다는 것을 가리킨다. 메소드 호출 명령은 메소드가 클래스 파일내에서 컴파일되기 때문에 바이트 코드가 된다.

메소드

```
void foo(Thread t, int i) {
    t.setPriority(i);
}
```

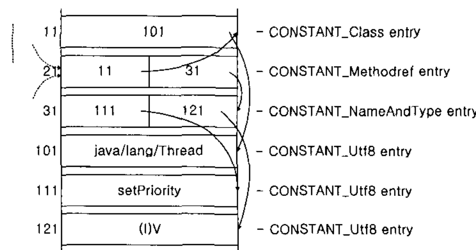
을 컴파일하면 다음과 같다.

```
Method public foo(Ljava/lang/Thread;I)v
  push Ljava/lang/Thread;I)v
  push I
  invokevirtual Thread.setPriority(I)V
```

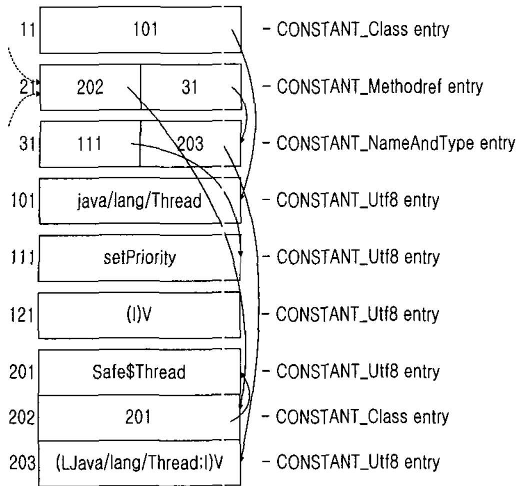
본 문은 Thread.setPriority(I)V를 보다 더 제한적인 메소드 Safe$Thred 클래스에 새로운 상한 값보다 더 높은 우선 순위를 갖고 애플릿을 허용하지 않는 Safe$ Thread.setPriority(Ljava/lang/ Thread;I)V를 부르는 메소드로 대체하기 위해서 메소드 레벨 수정을 한다. 인스턴스를 위하여, t.setPriority(5)는 Safe$Thread.setPriority(t,5)가 되므로, 새로운 메소드는 메소드 매개변수의 정수형으로 우선 순위를 가지기 때문에 메소드의 상한 값과 비교된다. 여기서 매개변수가 더 높다면, 매개변수가 상한 값으로 설정되므로 결국, 새로운 메소드는 수정된 매개변수 Thread.setPriority(I)V를 호출한다.

C. method에 의한 수정

클래스의 메소드 static method나 클래스 인스턴스의 메소드 instance method는 CONSTANT_Methodref로 상수풀 요소이다. 그림 8(a)에서 보여지는 것과 같이 CONSTANT_Methodref 요소는 메소드가 클래스의 멤버임을 나타내는 CONSTANT_Class 요소와 메소드의 이름과 정보를 CONSTANT_NameAndType 요소로 가리킨다. 본 연구에서는 CONSTANT_Class 요소와 CONSTANT_NameAndType 요소가 각각 java/lang/ Thread와 setPriority, (I)V CONSTANT_Uft8 요소로 수정되어 나타낸 것이다.



(a) Thread.setPriority(I)V에 의한 수정



(b) Safe$Thread.setPriority(Ljava/lang/Thread;I)V에 의한 수정

그림 9. 메소드 참조를 수정한, 메소드 레벨에 의한 수정

Fig. 9 Method-Level Modification Substitutes Method Reference

그림 9(b)에서 CONSTANT_Methodref 요소는 java/lang/Thread 클래스를 나타내는 이전의 CONSTANT_Class 요소 대신에 새로운 CONSTANT_Class 요소에 의해 수정된다. 메소드 정보가 변경되었기 때문에, 새로운 메소드 정보 (Ljava/lang/Thread ; I)V에 대한 기호적인 이름을 나타내는 CONSTANT_Uft8 요소가 추가된다. CONSTANT_NameAndType 요소가 메소드 정보에서 새로운 CONSTANT_Uft8 요소로 수정되면, 그 결과로 CONSTANT_Methodref 요소는 새로운 메소드 Safe$Thread.set Priority(Ljava/lang/Thread ; I)V로 수정된다.

D. method 호출 명령에 의한 수정

인스턴스 메소드 호출을 위한 invokevirtual과 클래스 메소드 호출을 위한 명령에서 그림 10와 같이 Safe$Thread.setPriority(Ljava/lang/Thread;I)V의 클래스 메소드 호출 invokestatic는 동일한 CONSTANT_Methodref 상수 풀 요소에 대한 인덱스를 매개변수로 갖지만, 오퍼랜드 스택은 서로 다르기 때문에 인스턴스 메소드 호출이 먼저 메소드에 속한 인스턴스 오퍼랜드 스택 등에 push함으로써 설정된다.

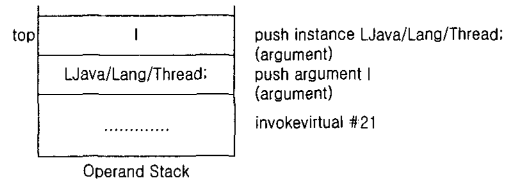


그림 10. 메소드 호출 명령을 위한 Operand 스택과 명령어 순서

Fig. 10 Operand Stack and instruction Sequences for methed in voking Instructions

새로운 메소드 Safe$ Thread. setPriority(Ljava /lang/Thread;I)V가 invokevirtual나 invokestatic로 변경함으로써 바이트 코드 프로그램에 추가할 수 있기 때문에 메소드 레벨 수정된다.

Ⅳ. 보안 제어 구현

A. 서비스 거부 공격 구현

본 장에서는 자바 바이트 코드 수정 기술을 이용하여 악성 애플릿의 공격을 막기 위한 대응책을 구현한다.

그림 11은 서비스 거부 공격에 의한 결과이므로 악성 애플릿은 "ACK!" 이름을 갖는 윈도우를 계속해서 생성하여 CPU 자원을 소모시켜 시스템을 다운시키는 것을 알 수 있다. 이러한 자원 소모 공격으로부터 보호하기 위해서는 윈도우 생성을 제어한다.

Frame은 final 클래스가 아니기 때문에 서브클래스 Safe$Frame이 생성될 수 있고, Safe$Frame은 모든 윈도우를 감시하고 제어한다. 클래스레벨 수정은 윈도우 크기와 윈도우 위치도 제한다.

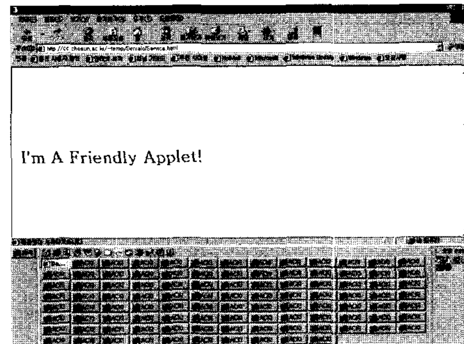


그림 11. 서비스 거부 공격 대응책 구현

Fig. 11 Denial of Service Attack

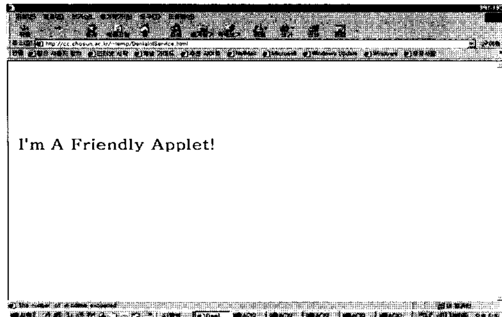


그림 12. 서비스 거부 공격 대한 대응 결과
Fig. 12 Result against Denial of Service Attack

그림 12에서 서비스 거부 공격에 대한 대응으로, Safe$Frame 클래스를 이용한 구현 결과로 윈도우 수를 5개로 제한하였으며, 5개 이상의 윈도우가 생성되려고 할 때에는 "The number of windows exceeded"라는 메시지를 화면에 나타내도록 하였다.

B. 위조 공격

생성자 메소드에 제한을 가하기 위해서, 생성자 메소드 호출이 JVM내에서 어떻게 실행되는가 알아진다.

JVM 클래스 인스턴스들은 JVM의 new 명령을 이용하여 클래스 인스턴스가 생성되고, 인스턴스 변수가 디폴트 값으로 초기화되면, 새로운 클래스 인스턴스(<init>)의 인스턴스 초기화 메소드가 다음과 같이 호출된다.

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

컴파일한 결과는 그림 13과 같이 invokespecial은 인스턴스 초기화 메소드 호출을 위한 자바 가상 머신 명령이고, 또한 superclass, private나 인스턴스 초기화 메소드와 같이 특별한 처리를 필요로 하는 인스턴스 메소드들을 호출한다.

```
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 13. create() 메소드의 컴파일 결과
Fig. 13 E-Mail Forging Attack

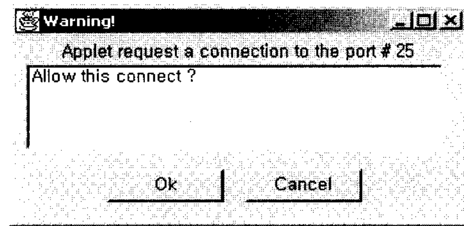


그림 14. 25번 포트로의 접속에 대한 경고창
Fig. 14 Warning Windows for connection to port 25

Socket은 브라우저내에서 final 클래스이므로, 메소드 레벨 수정을 통하여 메소드를 변경한다. 안전한 static 메소드 Safe$Socket.init는 클래스 메소드이기 때문에, 모든 소켓 연결을 감시하고 제어할 수 있다. Safe$Socket.init는 25번 포트 요청을 제외한 모든 요청에 대해 소켓 연결을 설정하고, 새로운 소켓 객체를 반환한다.

Safe$Socket.init에 의해 25번 포트로의 접속을 제어하기 때문에 그림 14과 같은 경고 창을 띄워 알리도록 하였다. 여기서 "OK" 버튼을 클릭하면 애플릿은 정해진 동작을 수행하고, 만약 "Cancel" 버튼을 클릭하면 25번 포트에 대한 접근을 거절하게된다. Safe$Socket.init는 Socket.<init> (Ljava/lang/String;I)V에 대한 Safe$ocket.init(Ljava/ang/tring;I)java/net/Socket;으로 제어된다.

최종적으로 수정된 코드는 그림 15와 같이, Safe$Socket.init는 static 메소드이므로 invokespecial은 invokestatic로 대체되며, 새로운 메소드가 소켓 객체를 반환하기 때문에, 스택으로부터 생성된 소켓 객체는 new 명령에 의해 제거된다.

```
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 15. 수정된 코드
Fig. 15 Modified Code

C. URL 위장 공격 제어

애플릿은 상태 표시줄에 위장된 URL을 표시함으로써 정보를 속일 수 있다. 이러한 위장 공

격은 표시된 URL과 실제로 로드된 웹 페이지의 URL간의 일관성을 검사함으로써 막을 수 있다.

그림16과 같이 Safe$ Applet Context. show Document가 호출되면, 먼저 URL 매개변수가 현재 상태 표시줄에 나타난 텍스트와 일치하는지 아닌지를 검사한다. 만약 일치한다면, 브라우저가 URL 매개변수에 의해 지정된 웹 페이지를 가져오도록 요청하고, 일치하지 않는다면 요청을 통과시키지 않고 경고 창을 띄워 제어한다.

AppletContext 인터페이스는 상속할 수 없기 때문에, 두 개의 인터페이스 메소드가 메소드 레벨 수정을 통하여도 제어된다. AppletContext. showStatus(S)V와 AppletContext.show Document(Ljava/net/URL;)V에 각각 Safe$ AppletContext.showStatus(S)V와 Safe$ AppletContext.showDocument(Ljava/net/URL;) V에 invokestatic 으로 제어되며, invokestatic 오퍼랜드는 nop 명령에 할당된다.

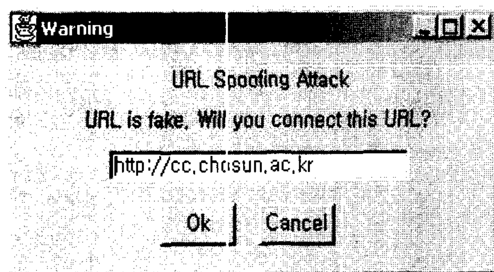


그림 16. URL 위장 공격에 대한 경고창
Fig. 16 Warring Windows for URL Spooltion Attack

D. 괴롭히는 사운드 공격 제어

괴롭힘 공격을 막기 위해서 사용자는 발생하는 사운드를 꺼야 하는데, 그 해결책은 모든 사운드 객체의 제어가 가능하도록 하는 것이다.

그림 17은 괴롭히는 사운드 공격을 나타낸 것으로 자바 라이브러리 인터페이스 AudioClip은 사운드를 연주하기 위한 중요한 메소드이므로 AppletContext.getAudioClip()과 Applet.getAudioClip() 이 인터페이스 객체를 반환한다. 객체의 loop 메소드는 루프내에서 소리의 연주가 시작하고, stop 메소드는 소리를 제어한다.

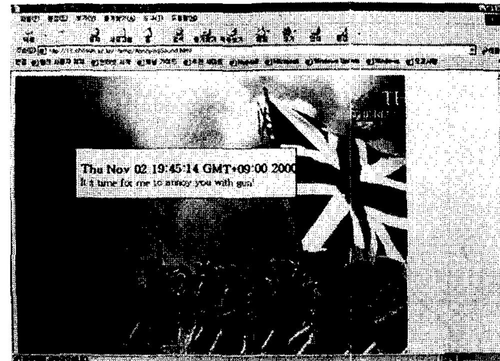


그림 17. 괴롭히는 사운드 공격
Fig. 17 Annoying Sound Attack

그림 18. 사운드 객체를 제어하기 위한 윈도우
Fig. 18 Windows for Controlling a Sound Object

애플릿이 객체의 loop 메소드를 호출할 때마다, 안전 메카니즘은 그림 18과 같이 사용자가 사운드를 끌 수 있는 윈도우를 띄워 사운드를 제어하도록 하였다.

AudioClip 연결은 상속할 수 없기 때문에, 안전 메카니즘은 메소드 레벨 수정을 이용하게되고, 거기에 대체해야 할 2개의 메소드가 있다. AudioClip.loop()V와 AudioClip.stop()V에 대해 Safe$AudioClip.loop()V와 Safe$AudioClip. stop() V에 제어 보안된다.

V. 결 론

자바 애플릿은 웹 서버에서 클라이언트의 브라우저로 전송할 때, 자바 애플릿이 악용될 경우 정보의 시스템을 파괴하거나, 시스템을 다운시키고, 정보 시스템에 유해한 공격을 유발시키는 등의 보안상의 문제를 일으키는 것을 제어할 수 있게 연구하였다.

본 논문에서는 악성 애플릿에 의한 공격 중에서 서비스 거부 공격에는 클래스 레벨 수정을 이용해 생성되는 윈도우의 수를 제한하여

공격에 대한 대응책을 구현하였고, 전자메일 위조 공격에도 메소드 레벨 수정을 이용해 웹 서버의 25번 포트에 접근 제어하는 방법을 통해 대응하였다. 또한 URL 위장 공격에는 메소드 레벨 수정을 이용하여 표시된 URL과 실제 로드된 웹 페이지의 URL 간의 일관성을 검사하여 공격에 대응하으며, 사운드를 이용한 괴롭힘 공격도 메소드 레벨 수정을 이용하여 사운드의 발생을 제어함으로서 공격을 차단하였다.

바이트 코드 수정 기술을 통해 악성 애플릿을 차단하기 위하여 웹 서버와 클라이언트 사이에 프록시 서버를 설치하여 악성 애플릿들을 프록시 서버 내에서 수정되도록 하였다.

본 논문에서는 인터넷 통신에서의 악성 애플릿들에 의한 보안 문제에 대하여 제안한 자바 바이트 코드 수정을 통해 악성 애플릿들의 공격을 제어한다.

참 고 문 헌

[1] J. Gosling, B. Joy, and G. Steele, The Java Language Specification, Menlo Park, Calif.: Addison-Wesley, 1996.

[2] T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Menlo Park, Calif.: Addison-Wesley, 1997.

[3] Li Gong, "Java Security Architecture(JDK 1.2)", Sun Microsystems, 1998

[4] J. Gosling, "The Java Language Environment", Sun Microsystems, 1996

[5] J. Steven Fritzinger, Marianne Mueller, "Java Security", Sun Microsytems, 1997

[6] Joseph A. Bank, "Java Security", http://swissnet.ai.mit.edu/~jbank/javapaper/javapaper.html

[7] Li Gong, R. Schemers, "Implementing Protection Domains in the Java Development Kit 1.2", Proceedings of Internet Society Symposium on Network and Distributed System Security: pp. 103-112, 1997

[8] Dahlia Malkhi, Michael Reiter, Avi Rubin, "Secure Execution of Java Applets using a Remote Playground"

[9] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson Modulak, "Language Definition". SIGPLAN Notices, 27(8), August 1992.

[10] G.C. Necula and Peter Lee, "Safe kernel extensions with runtime checking", In Proceedings of the 2nd Symposium on Operating Systems Design and Implementation, October 1996.

나 상 동(Sang-Dong Ra) 정회원

1968년 : 조선대학교 전기공학과 졸업(공학사)
1980년 : 건국대학교 대학원 졸업(공학석사)
1995년 : 원광대학교 대학원 졸업(공학박사)
1995년~1996년 : Dept. of Electrical & Computer Eng. Univ. of California Irvine 연구 교수.
1998년 : 조선대학교 전자계산소 소장 역임
1973년~현재 : 조선대학교 컴퓨터공학부 교수
2001년~2002년: Dept. of Electrical & Computer Eng. Univ. of California Irvine 연구교수

<주관심분야> 실시간 통신, 디지털 통신망 , 데이터 및 이동통신, TOM, 적응 신호처리 등.

김 문 환(Moon-Hwan Kim) 정회원

1988년 : 한국방송통신대학교 전자계산학과 졸업(이학사)
1991년 : 조선 대학교 산업대학원 전자 계산 전공 졸업(공학석사)
1983.8~1993.11 : KT통신망본부근무
1993.12~1997.1 : KT연수원 전송학부 전임교수
1997.2~2003.4 : 현재 KTF 광주사업본부 교환운용팀