보안 취약점 검사를 위한 AOP 기반의 동적 분석
(Dynamic Analysis based on AOP for Checking Security Vulnerability)

서 광 익 † 최 은 만 ††
(Kwangik Seo) (Eun Man Choi)

요 약 국제 웹 어플리케이션 보안 연구 단체(OWASP)는 2007년에 이어 2010년에도 취약점 대부분이 사용자의 외부 데이터 입력에 기인한 것으로 발표했다. 이러한 오염된 입력 데이터는 실행 시점에서 결정되기 때문에 동적인 취약점 분석이 필요하다. 동적 분석 방법은 주로 실행 시점의 데이터 분석이나 경로 흐름 분석을 위해 인스트루먼트를 코드 내에 삽입 한다. 하지만 직접적인 코드의 삽입은 관리와 확장이 어려워 분석 범위와 대상이 증가할 때 마다 코드 조각들이 흩어지게 된다. 게다가 인스트루먼트 모듈과 시험 대상 모듈 간의 결합도가 높아진다. 따라서 개발이나 유지보수 단계에서 삽입한 분석 코드를 수정하거나 확장하는데 많은 노력이 필요하게 된다. 본 논문은 이러한 문제점을 해결하기 위해 AOP를 이용하여 취약점을 하나의 관심사로 분류함으로써 결합도의 증가 없이 삽입, 삭제와 유연한 확장이 용이한 방법을 제안한다.
키워드 : 시큐어 코딩, 코드 취약성, 보안, 동적 분석, AOP

Abstract OWASP announced most of vulnerabilities result from the data injection by user in 2010 after 2007. Because the contaminated input data is determined at runtime, those data should be checked dynamically. To analyze data and its flow at runtime, dynamic analysis method usually inserts instrument into source code. Immediate code insertion makes it difficult to manage and extend the code so that the instrument code would be spreaded out according to increase of analysis coverage and volume of code under analysis. In addition, the coupling gets strong between instrument modules and target modules. Therefore developers will struggle against modify or extend the analysis code as instrument. To solve these problem, this paper defines vulnerabilities as a concern using AOP, and suggest the flexible and extensible analysis method to insertion and deletion without increase of coupling.
Key words : Secure Coding, Code Vulnerability, Security, Dynamic Analysis, AOP

1. 서 론

네트워크와 인터넷의 발달로 인해 모든 단말기와 소프트웨어가 통신하는 환경에서 소프트웨어 취약점을 이용한 공격이 급증하고 있다. Gartner의 사이버 위협의 동향 보고서에 의하면 어플리케이션 취약점을 이용한 공격이 75%에 이르고 있다[1]. 이러한 보고는 소프트웨어의 취약점을 점검하고 개선하여 안전한 시스템의 구축이 시급함을 말한다.

보안을 위한 취약점을 점검하는 방법은 크게 정적 분석 방법과 동적 분석 방법이 있다. 정적 분석 방법은 애플리케이션을 실행시키지 않고 소스코드 자체를 분석한다[2]. 특히 소프트웨어를 개발하는 단계에서 주로 사용하여 취약점의 근본적인 원인 정보를 제공한다. 하지만 정적 분석은 실행 중인 프로그램에 대한 분석이 아니기 때문에 실행 전 취약점의 위치나 특성을 알아낼 수 있

† 학생회원 : 동국대학교 컴퓨터공학과 연구원
bradseo@dongguk.edu
†† 종신회원 : 동국대학교 컴퓨터공학과 교수
emchoi@dgu.ac.kr
논문접수 : 2010년 7월 8일
심사완료 : 2010년 8월 31일

Copyright©2010 한국정보과학회 : 개인 목적이나 교육 목적인 경우, 이 저작물의 전체 또는 일부에 대한 복사본 혹은 디지털 사본의 제작을 허가합니다. 이 때, 사본은 상업적 수단으로 사용할 수 없으며 첫 페이지에 본 문구와 출처를 반드시 명시해야 합니다. 이 외의 목적으로 복제, 배포, 출판, 전송 등 모든 유형의 사용행위를 하는 경우에 대하여는 사전에 허가를 얻고 비용을 지불해야 합니다.
정보과학회논문지: 소프트웨어 및 응용 제37권 제10호(2010.10)

다. 하지만 실시간으로 취약점 검사를 위해서는 프로그램을 실행해야하는 동적 분석 방법을 바람직하다. 동적 분석은 주로 실행 중인 프로그램을 모니터링하거나 성능을 평가하기 위해 사용한다[3]. 실행중인 소프트웨어에 대한 분석이기 때문에 취약점을 실행 정황에 근거해 찾아 낼 수가 있다. 동적 분석은 정적 분석에 비해 정확도가 높은 반면 분석 기능이 있는 코드를 기존 코드에 삽입하거나 삭제하는데 많은 노력이 요구된다. 또한 동적 분석은 분석하고자 하는 취약점에 따라 분석 방법이나 규칙이 달라져야 한다. 따라서 취약점 분석을 위해 취약점의 종류와 특성에 맞도록 유연하면서 확장성이 좋은 분석 방법이 필요하다.

본 논문은 보안 취약점을 분석하는데 있어 확장성이 높은 AOP(Aspect-Oriented Programming)를 이용한 동적 분석 방법을 제안한다. AOP는 사용자의 핵심 요구사항을 핵심 모듈과 별개로 보조적인 횡단 모듈을 분리할 수 있다[4]. 이러한 기능은 기존 시스템의 구조에 영향을 미치지 않고 분리 개발 및 기능 추가가 가능하다. 그림 1은 핵심 관심과 횡단 관심의 예이다. 핵심 관심은 계좌이체와 입출금 그리고 이자계산과 같은 주요 핵심 기능이나 모듈이다. 횡단 관심은 모든 핵심 관심에 필요한 로깅, 보안, 트랜잭션이 각 핵심 모듈에 횡단으로 흩어져 있다. 이러한 경우 AOP는 계좌이체, 입출금, 이자계산을 중심으로 모듈화 되어 있는 시스템을 로깅과 보안 그리고 트랜잭션을 구현하고 있는 코드부분을 횡단으로 분리할 수 있다. 본 논문은 동적 취약점을 횡단관심으로 정의하고 흩어져있는 취약점을 동적으로 분석할 수 있도록 AOP를 적용한 방법을 제안한다.

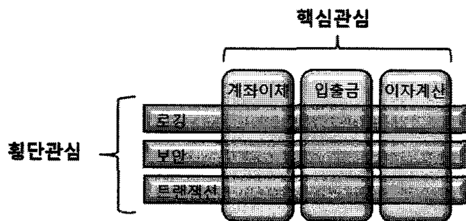

그림 1 횡단 관심사(cross-cutting concern)

2. 동적 취약점 검사를 위한 AOP

최근 프로그램 코드에 잠재된 취약점 분석에 대한 관심이 높아지고 있다[5]. 취약점 분석은 외부의 공격에 대한 안전성 확보 여부를 검사하는 안전한 프로그래밍(Secure Programming)을 위해 정적 기법을 이용할 수 있다[6]. 정적 분석은 커버리지는 높지만 정확도가 낮다. 반면 동적 분석은 정확도가 높다. 비록 동적 분석은 실행중인 프로그램을 분석하기 때문에 많은 비용이 들지만 높은 정확도를 위해서는 반드시 필요한 분석 방법이다.

동적 분석은 사용자 입력 값을 즉시 분석하여 데이터에 특수 문자열(escape sequence)나 불필요한 문자열이 있는지 알려준다. 대부분 입력 값 검사를 위해서 기존에 개발되어 있는 코드에 인스트루먼트(instrument)를 삽입 한다. 인스트루먼트 삽입 자체는 문제없는 프로그램에 또 다른 결함이나 취약점이 될 수도 있다. 또한 인스트루먼트의 코드 양이 많거나 여러 곳에 산재된다면 인스트루먼트를 관리하는 것만으로도 많은 추가비용과 부작용이 발생한다.

AOP는 핵심 관심에 산재되어 있는 횡단 관심을 분리하여 다룰 수 있는 방법을 제공한다[4]. 동적검사는 횡단 관심로 구분할 수 있다. AOP 방법을 적용하여 동적검사를 구현하면 기존 코드와는 독립적인 코드가 된다. 따라서 기존 코드와의 결합도가 발생하지 않는다. 또한 여러 코드 내에 흩어져 있는 인스트루먼트를 애스팩트에 캡슐화 할 수 있다. 이는 인스트루먼트의 삽입, 추가, 삭제 과정에서 기존 코드를 수정할 필요가 없어 이에 대한 부작용을 방지할 수 있다.

3. CERT의 자바 취약점

OWASP(Open Web Application Security Project)는 가장 높은 순위의 보안 취약점의 원인을 외부의 데이터 주입(Injection)인 것으로 발표했다[7]. CERT의 자바 취약점 중 10. Input Validation and Data Sanitization(이하 10. IDS라 함)은 OWASP에서 발표한 데이터 주입과 관련된 취약점과 예방 지침을 제시하고 있다[5]. 따라서 본 논문은 10. IDS를 근간으로 외부의 데이터 주입에 대한 취약점을 정의하고, AOP를 적용하여 이를 분석한다.

10. IDS의 취약점과 취약점 발생 가능 위치를 표 1에 정리했다. 표 1을 보면 다양한 취약점이 있지만 검사 지점으로 고려해야 하는 위치는 대부분 메소드의 입력 지점이라는 것을 알 수 있다. 취약점 발생 가능한 부분은 사용자 정의 메소드 또는 라이브러리 메소드 자체를 말하지만, 그 메소드가 다루고 있는 리턴 값이나 파라미터도 포함한다. 따라서 10. IDS에 의하면 입출력 취약점은 크게 두 가지로 나눌 수 있다. 첫째는 취약성 발생이 가능한 메소드이다. 이러한 메소드의 집합을 본 논문에서는 메소드 블랙리스트라 정의한다. 둘째는 취약성을 일으키는 원인이 되는 오염된 문자(열)의 입력이나 출력의 경우라 할 수 있다. 본 논문에서는 이러한 문자(열)의 집합을 문자(열) 블랙리스트라 한다. 본 논문은 메소드 블랙리스트나 문자(열) 블랙리스트를 정의하고 블랙리스트가 사용되거나 또는 호출된 시점을 동적으로 분석할 수 있는 애스팩트를 자동 생성한다.

표 1 10. IDS 입출력 취약점

식별자	취약점	검사 가이드	취약점 발생 가능 위치
IDS00-J	검증 안 된 입력 값	입력 값 검사	UI 입력 메소드
IDS01-J	불필요 문자 입력	입력 값 검사 및 문자 제거	UI 입력 메소드
IDS02-J	문자열 처리와 정규화 순서	정규화 후 문자열 처리	정규화 메소드
IDS03-J	불필요 문자 제거	비문자(noncharacter) 제거 함수 검사	문자 제거 메소드
IDS04-J	보호 대상 정보 검사 없이 출력	출력 내용 보안성 확보	출력 메소드
IDS05-J	라이브러리 메소드 간 파라미터 검사	메소드 파라미터 검사	파라미터 있는 메소드
IDS06-J	OS 명령어 실행	시스템 명령 메소드 검사	시스템 메소드
IDS07-J	SQL 주입	입력 값 검사	UI 입력 메소드
IDS08-J	XML 주입	입력 값 검사	UI 입력 메소드
IDS09-J	XPath 주입	입력 값 검사	UI 입력 메소드
IDS10-J	XML 외부 파일 사용	외부 파일 내용 검사	파일 읽기 메소드
IDS11-J	디렉토리 접근	디렉토리 접근 특수 문자 검사	디렉토리 접근 메소드
IDS12-J	코드 삽입	스크립트 엔진의 실행 메소드 파라미터 검사	UI 입력 메소드
IDS13-J	국제 코드에 따른 문자 조합	문자 조합 메소드 검사	문자 변환 메소드
IDS14-J	(유니코드↔바이트)문자열 변환 시 원시 정보 손실	문자열 처리 메소드 검사	문자열 변환 메소드
IDS15-J	URL 파싱 후 접근	URL 접근 검사	URL 생성자
IDS16-J	국가/지역적 특화 메소드 사용	문자(열) 처리 메소드 검사	String.toLowerCase() 또는 String.toUpper()
IDS17-J	Pattern 인스턴스의 특수 문자 패턴 검색	패턴 매소드 검사	Pattern.split()

```
String method(String str) {           //① 파라메타 str로 입력
name ← extern_method( );              //② 오염된 데이터 ;--
str ← "SELECT
AccountNumber FROM
Users " & _                            //③ 취약성 있는 문자열
           "WHERE                      //name = ";--"
Username='" & name & "'
AND Password='" &
password & "'";
return str;                           //④ 오염된 데이터 출력
}
```

그림 2 10. IDS의 입출력 취약성 예

표 1의 구체적인 취약점의 예를 그림 2에서 볼 수 있다. 첫 번째로 method()가 str의 값을 전달 받는 경우이다. 그리고 exten_method()는 값을 리턴하는 메소드인데, 실행 결과 "② 오염된 데이터"를 반환하는 경우이다. 그림 2에서는 ";--"를 반환하면 뒤의 문장인 비밀번호 필드에는 아무 값이나 입력한 후 로그인을 시도하여 취약점이 발생한다. 이와 같이 취약점은 IDS.10과 같이 대부분 입출력에 의해 유입된다.

4. 동적 취약점 분석

본 논문에서 제안하는 동적 분석 방법은 그림 3과 같이 크게 세 단계로 이루어진다. 첫 번째 단계는 애스팩트 생성기가 추상 애스팩트를 확장하여 검사 대상 시스템에 직조될 취약점 분석 애스팩트를 자동 생성한다. 두 번째 단계에서는 컴파일하여 애스팩트와 분석 대상 클래스를 직조한다. 세 번째 단계는 추상 애스팩트에 정의되어 있는 모니터링과 로깅 모듈이 실행 시간에 취약점을 분석하고 결과를 출력한다.

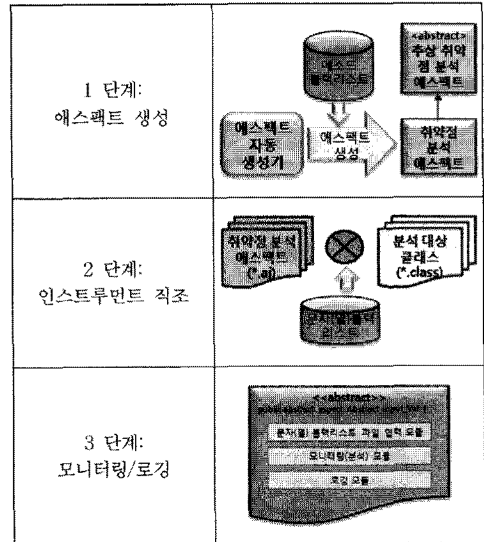

그림 3 동적 취약점 분석

4.1 애스팩트 자동 생성

애스팩트 자동 생성은 그림 3의 첫 번째 단계에 해당한다. 애스팩트 자동 생성기는 메소드 블랙리스트를 읽고 새로운 취약점 분석 애스팩트를 생성한다. 본 논문의 예에는 Java의 스윙(Swing)을 이용한 입력 값을 검사한다. JTextField, JPasswordField 또는 JTextArea 클래스에 속한 getText() 메소드나 getPassword()등이 C:\\mthd_blacklist.txt에 리스트로 정의 되어 있다. 그림 4의 AG 클래스는 블랙 리스트를 fis로 불러와서 그림 5

```
public class AG {
                    ------ 중략 -------
fis = new FileInputStream("C:\\mthd_blacklist.txt");//메소드 블랙리스트 파일
isr = new InputStreamReader(fis);
br = new BufferedReader(isr);

FileOutputStream fos=null;
BufferedOutputStream bos=null;

try{
fos=new FileOutputStream("C:\\input_Check.aj");//분석 애스팩트
bos = new BufferedOutputStream(fos);
String str = "public aspect Input_Check extends Abstract_input_Vul\n" + "{\n";
        // Abstrac_input_vul 추상 메소드 상속한 Input_Check 애스팩트
bos.write(str.getBytes());
bos.flush();
                    ------- 중략 -------
while((char_blacklist = br.readLine()) != null){ //메소드 블랙리스트 쓰기
bos.write((char_blacklist.getBytes()));
bos.flush();
bos.write("&&".getBytes());
bos.flush(); }
                    -------- 후략 --------
```

그림 4 애스팩트 생성 클래스

```
public aspect Input_Check extends Abstract_input_Vul
{
pointcut methodBlacklist(String input) :
        call (* JTextField.getText()) && (* JPasswordField.getPassword())
                && (* JTextArea.getText());
}
```

그림 5 자동 생성된 애스팩트

```
public abstract aspect Abstract_input_Vul {
                    ------ 중략 ------
declare warning : methodBlacklist(String input) : "attack";//취약점 경고 출력
                    ------ 중략 ------
after() returning(String input): methodBlacklist() //충고
{
  inputBlacklist(); // 블랙리스트 읽기 모듈 호출
  ArrayList<Object> object = new ArrayList<Object>();
  Pattern pattern = Pattern.compile(char_blacklist); //검사를 위한 문자(열) 패턴
  monitor(input, pattern); //모니터링 모듈 호출
  object.add(thisJoinPoint.getThis());
  object.add(thisJoinPoint.getSourceLocation());
  object.add(thisJoinPoint.getSignature());
  object.add(thisJoinPoint.getTarget());
  logger(object); //로깅 모듈 호출
}
                    ------ 중략 ----

void inputBlacklist(){ //문자(열) 블랙리스트 읽기 모듈
 try{
 fis = new FileInputStream("C:\\char_blacklist.txt");
 isr = new InputStreamReader(fis);
                    ------ 중략 ------
}
void monitor(String input, Pattern pattern) //모니터링 모듈
{
  System.out.println("<<Aspect>>input: "+ input);
  if(pattern.matcher(input).find()){
  System.out.println("invaild input : " + pattern.matcher(input).toString());
  }
  if(input.length()> 8 || input.length()==0 || input.indexOf("\'")!=-1)
  System.out.println("attampted attack");
}

void logger(ArrayList Object_this){ //로깅 모듈
    System.out.println("vulnerable hotspot\n");
    System.out.println("Dynamic thisJoinPoint: " + Object_this.get(0));
    System.out.println("Static thisJoinPoint: " + Object_this.get(1));
    System.out.println("Static thisJoinPoint: " + Object_this.get(2));
}
}
```

그림 6 추상 애스팩트

와 같이 애스팩트를 생성할 때 교차점으로 정의한다. 즉 그림 3의 애스팩트 생성 클래스의 실행 결과 그림 6의 애스팩트를 확장한 그림 5의 애스팩트가 생성된다. 그림 6의 추상 애스팩트는 메소드 블랙리스트를 입력받은 후, 모니터링하고 로깅하는 메소드를 포함하고 있다. 따라서 그림 6를 확장하거나 오버라이딩하여 검사 규칙을 추가 및 수정할 수도 있다.

4.2 애스팩트와 클래스의 직조

그림 3의 두 번째 단계는 취약점 검사 애스팩트와 분석 대상 클래스가 직조되는 단계이다. 애스팩트 컴파일러를 이용하여 컴파일하면 애스팩트는 분석 대상 클래스와 직조된다. 컴파일하는 과정에서 취약점 검사가 필요한 위치를 그림 7과 같이 미리 보여준다. 만약 메소드 블랙리스트나 문자(열) 블랙리스트에 정의된 항목이 검사 대상 클래스 내에 있다면 그 위치를 알려준다. 그림 7과 같은 예비 정보는 프로그램을 실행하기 전에 위치를 추적하여 취약점에 대한 대비 여부를 개발자가 확인할 수 있도록 한다.

Description	Resource	Path	Location
Warnings (13 items)			
attack	IDS07_SQLInjection.java	/Analysis_withAOP/src	line 57
attack	IDS_06_CommandInjection.java	/IDS/src	line 45
attack	IDS_07_SQLInjection.java	/IDS/src	line 40
attack	TextFieldTest.java	/Analysis_withAOP/src	line 45
attack	TextFieldTest.java	/IDS/src	line 47
attack	TextFieldTest.java	/IDS/src	line 48
The import java.io	IO_Vunerabili -- 중략 --	/Analysis_withAOP/src/ID...	line 1

그림 7 컴파일 시점에서의 취약점 점검

4.3 모니터링과 로깅

그림 3의 세 번째 단계는 실행 시점에서 취약점을 모니터링하고 로깅한다. 그림 6의 after() 충고가 실행되면 추상 애스팩트에 정의된 모듈이 실행된다. 파일에서 읽어온 문자(열) 블랙리스트를 Pattern.compile()의 파라미터로 정의한다. IDS06, IDS08, IDS09, IDS10에 의하면 SQL과 XML 표현과 관련된 < 또는 >와 같은 문자는 입력 금지 문자이다. 또한 IDS07과 같이 디렉토리 접근을 위한 콜론(:)이나 \와 같은 문자도 마찬가지이다. 문자열 블랙리스트 읽기 모듈은 "C:\\char_blacklist.txt"에 정의된 문자(열)을 정의하고 있다. 추상 애스팩트는 이를 fis로 선언하여 문자 블랙리스트를 참조한다.

실시간 모니터링 모듈은 사용자가 입력한 데이터를 Pattern과 대조하여 금지된 문자(열)이 입력되거나 정의한 입력 형식에서 벗어나는지 실행 시점에서 실시간으로 검사한다.

그림 9는 그림 8의 예제 클래스를 모니터링하고 로깅한 결과이다. 그림 8의 TextFieldTest 클래스의 48번째 줄에 있는 getText() 메소드의 입력값에 대한 분석 결과이다. 사용자는 '*OR 1=1* --와 같이 SQL 주입 공격을 시도를 모니터링하여 공격을 시도하고 있다는 정보를 알려준다.

5. AOP를 이용한 인스트루먼트 효과

본 연구에서 제안한 방법의 특징을 표 2에 정리했다.

```
5   public class TextFieldTest implements ActionListener {
                              ------- 중략 -------
47  public void actionPerformed(ActionEvent ev){
48  System.out.println("ID:" + input_id.getText() + "₩n");
49  System.out.println("PW:" + input_pw.getPassword().toString()+ "₩n");
                              ------- 중략 -------
72  }
```

그림 8 취약성이 있는 코드

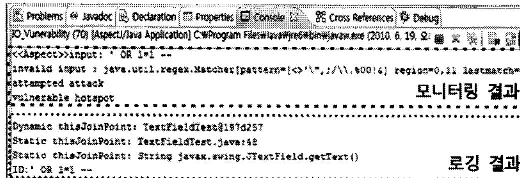

그림 9 실시간 모니터링 및 로깅

표 2 시점에 따른 분석 내용

	컴파일 시점	사용자 입력시점	제어 로직 실행 시점
대상	소스 코드	입력 값	실행 메소드 또는 파라메터
기법	취약점 분석	동적 분석	영역별 동적 분석 또는 정적 분석
검사 자료	블랙리스트	블랙리스트	블랙리스트 또는 메소드 실행 규칙

표 2는 분석 시점과 대상에 따라 분석 방법과 기준을 자세히 보이고 있다. 컴파일 시점에서는 소스 코드를 대상으로 취약점을 분석하는 것과 같은 정적 분석을 한다. 그리고 입력 행위 시점에서의 취약점 검사를 동적으로 분석한다. 분석의 기준은 블랙리스트를 이용했다.

본 절에서는 표 3과 같이 GUI 기반 공개 소스를 이용하여 취약점을 찾아 보고, 그에 대한 검사 모듈이 있는지 분석했다. LoanCalculator[8]는 IDS00에 해당하는 세 개의 취약점이 존재하고 이를 검사하는 모듈 한 개가 있다. ReadURL[9]은 취약점 IDS00과 IDS15에 해당하고 각각 한 개씩 취약점이 있으나, 이를 검사하는 코드는 모듈화되어 있지 않고 기능을 구현하는 코드 안에 흩어져 있다. MgtFile[10]과 Flight_RSV[11]는 검사 코드가 모듈화 되어있거나 코드 안에 흩어져 있다. 일반적으로 취약점을 검사할 경우 검사하고자 하는 기능 다음에 삽입한다. 예를 들면 ReadULR[9] 어플리케이션의 ReadURLApplet.java 파일의 137번째 줄에 있는 URL url = new URL(getDocumentBase(), urlName);은 URL의 주소를 검사 없이 저장하고 있어 IDS15-J에 해당하는 취약점을 가지고 있다. 또한 Flight_RSV[11]의 Login.java의 172번 째 줄은 IDS00에 해당하는 취약점을 가지고 있는 sCheck=type.TFUserName.getText();이다. 하지만 Login.java에서는 이 취약점을 검사하기 위한 코드는 모듈화 되어 있지 않고 172번 째 줄 이하에

표 3 취약점 검사 코드 모듈화 비교

	전체 코드 라인 수	관련 취약점	찾은 취약점 개수	취약점 검사 모듈화 여부	검사 모듈의 라인수
LoanCalculator[9]	420	IDS00	3	1	42
ReadURL[9]	219	IDS00	1	없음	7
		IDS15	1	없음	0
MgtFile[10]	1942	IDS00	1	1	152
		IDS11	2	없음	4
Flight_RSV[11]	1031	IDS00	4	2	10
		IDS11	4	없음	0

if-else 문으로 단순히 삽입되어 있어 검사 규칙이나 172번 째 줄을 수정하게 되면 취약점 검사 코드의 영향이 커지게 된다. 이와 같이 검사 코드를 삽입한 경우는 검사 코드에 대한 관리가 어렵고 중복이 발생할 수 밖에 없다. 검사 코드를 모듈로 정의한 경우도 검사 규칙을 수정하면 그에 대한 파급효과를 예상해야 한다. 또한 모듈을 호출하는 코드가 필요하기 때문에 개발자가 직접 코드를 추가해야 한다. 하지만 AOP를 이용한 경우 직조 과정을 통해 원본 코드를 수정하지 않으면서 검사 인스트루먼트를 자동으로 삽입한다. 또한 한 번의 교차점(pointcut)과 충고(advice)의 정의로 여러 곳에 분산되어 있는 사용자 입력에 대한 취약점을 모두 포착할 수 있어서 산만한 인스트루먼트 삽입이나 수정에 대한 부작용을 방지할 수 있다.

6. 관련 연구와의 차이점

그 동안 보안성을 높이기 위해 AOP를 적용한 연구가 활발히 진행되었다[12,13]. 특히 주요 기능을 구현하고 있는 모듈에 조각조각 흩어져있는 보안 측면의 코드를 애스팩트로 슬라이싱하여 추적하거나 또는 유지보수 단계에서 보안 모듈을 이미 개발되어 있는 코드에 추가하는데 있다. 하지만 본 논문은 프로그램 코드가 가지고 있는 취약성을 분석하는데 목적이 있다. 따라서 본 논문은 블랙리스트에 있는 메소드의 호출 시점에 대한 분석 뿐 만 아니라 메소드나 문자(열)에 대한 블랙리스트를 검사하는 애스팩트를 자동으로 구현하는 것이 큰 차이점이라 할 수 있다.

7. 결 론

본 논문에서는 취약점 분석을 위한 애스팩트를 자동으로 생성하는 방법을 제안했는데, 크게 네 가지로 정리할 수 있다. 첫째, 애스팩트 컴파일 시점에서 취약점을 검색하여 개발자가 사전에 취약점을 점검할 수 있는 방

법을 마련했다 둘째, 실행시점에서 취약점 모니터링과 로깅이 가능한 확장성 있는 애스팩트를 정의했다. 이러한 애스팩트를 이용한 취약점 분석은 기존의 검사 인스트루먼트와는 전혀 다른 접근 방법이다. 원본 코드에 검사 코드를 삽입하거나 수정할 필요가 전혀 없다. 셋째, 애스팩트를 자동으로 생성할 수 있는 방법을 간단히 소개 했다. 넷째, 검사 규칙이나 기준을 독립적으로 정의하도록 하여 확장성을 높였다.

참 고 문 헌

[1] Gartner, "Now is the time for Security at Application Level," www.gartner.com/DisplayDocument?id=487227, 2006, 12, 1
[2] D. Binkley, "Source Code Analysis: A Road Map," *2007 Future of Software Engineering(FOSE07)*, pp.104-119, 2007.
[3] D. D. Cruz, P. Henriques, and J. S. Pinto, "Code Analysis: Past and Present," *Proceedings of the Third International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert2009)*, 2009.
[4] Kiczales, Gregor, G Lamping, A Mendhekar, C Maeda, C Lopes, J Loingtier, J Irwin, Aspect-Oriented Programming, *Proc. of ECOOP*, vol.1241, pp.220-242, 1997.
[5] CERT Secure Coding Standards, http://www.securecoding.cert.org
[6] Brian Chess, "Secure Programming with Static Analysis," Addison-Wesley Professional, 2007.
[7] Open Web Application Security Project, "The Ten Most Critical Web Application Security Vulnerabilities," http://www.owasp.org/images/0/0f/OWASP_T10_-_2010_rc1.pdf
[8] LoanCalculator project, http://www.planet-source-code.com/vb/scripts/BrowseCategoryOrSearchResults.asp?lngWId=2&txtCriteria=calcul
[9] ReadURL project, http://www.faqs.org/docs/javap/source/index.html
[10] File Management project, http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=5441&lngWId=2
[11] Flight Reservation System project, http://www.planet-source-code.com/vb/scripts/ShowCode.asp?txtCodeId=4237&lngWId=2
[12] Li Yuan-yuan, "AOP-Based Attack on Obfuscated Java Code," *IEEE 2009 International Conference on Computational Intelligence and Security*, pp.238-241, 2009.
[13] Lionel, LaurenceDuchien, Roberto, GabrielHermosillo. "Using Applications," *Journal of Software*, vol.2, no.6, pp.53-63, December 2007.

서 광 익

2002년 동국대학교 컴퓨터공학과 공학사 2004년 동국대학교 컴퓨터공학과 공학석사. 2009년 동국대학교 컴퓨터공학과 공학박사. 2005년~2010년 동국대학교 외래강사. 2010년~현재 (주)STA컨설팅 책임연구원. 관심분야는 소프트웨어 테스트, 소프트웨어 품질, 프로세스, Traceability, 소프트웨어 취약성

최 은 만

1982년 동국대학교 전산학과 졸업(학사) 1985년 한국과학기술원 전산학과(공학석사). 1993년 일리노이 공대 전산학과(공학박사). 1985년~1988년 한국표준연구소 연구원. 1988년~1989년 데이콤 주임연구원. 1992년 시카고 주립대 전산학과 강사. 1997년~2004년 한국정보과학회 소프트웨어공학 연구회 운영위원. 2001년~2005년 한국정보처리학회 학회지 편집위원. 2002년 카네기 멜론 대학 소프트웨어공학 단기 과정 연수. 2000년, 2007년 콜로라도 주립대 전산학과 방문교수. 1993년~현재 동국대학교 컴퓨터공학과 교수. 관심분야는 객체지향 설계, 소프트웨어 테스트, 프로세스와 메트릭, Program Comprehension, AOP