Springframework messaging 원격 코드 실행 취약점

(CVE-2018-1270)
분석보고서

2018. 11. 19

Writer : 정수림

1. 개요

Spring Framework는 공공기관의 웹 서비스 개발 시 쓰이고 있는 동적인 웹 사이트를 개발하기 위한 여러 가지 서비스를 제공하고 있다. CVE-2018-1270 취약점은 공격자가 조작된 메시지를 서버에 전송하여 악의적인 명령이 실행되도록 한다. 서버는 공격자의 요청을 처리하여 원격 쉘을 제공하고, 이를 통해 서버 내의 중요정보들을 탈취하게 된다. 해당 취약점은 현재 보안 패치 및 조치 가이드가 제공된 상태이지만 공격방법이 매우 쉬우며 공격을 받을 시 시스템에 미치는 영향도가 크기 때문에 취약점에 대해 분석해 보았다.

2. 영향 받는 소프트웨어

CVE	취약한 버전	위험도
CVE-2018-1270	4.3 - 4.3.15 5.0 - 5.0.4	high

3. 취약점 원리

Spring framework에서는 대화형 웹 서비스를 위해 'spring-messaging' 모듈을 사용하는데, 이때 STOMP(Simple Text-Oriented Messaging Protocol) 을 이용한 'Web Socket' 연결 시 공격자가 삽입한 악성코드가 서버에서 실행되면서 취약점이 발생한다. 구체적으로는 공격자가 서버의 특정 destination으로부터 메시지 구독을 요청할 때 악성코드를 삽입한 후, SEND 명령을 통해 응용프로그램 destination으로 문자열을 전달하면 broker에서 클라이언트 정보를 획득하는 과정에 발생한다.

- addSubscriptionInternal() 함수

공격자가 악성코드를 삽입하여 SUBSCRIBE 명령을 통해 메시지를 구독받고자 하는 destination을 전송하면 서버에서는 다음과 같은 과정을 거쳐 실행하게 된다. 먼저 클라이언트가 subscribe를 요청하면, 서버 안에 addSubscriptionInternal() 함수에서 요청 내 헤더 정보를 처리한다.

경로 : gs-messaging- stomp - websocket[boot] ->Maven Dependencies -> Spring-messaging-5.0.4.RELEASE.jar -> org.springframework.messaging.broker -> DefaultSubscriptionRegistry.class

```
protected void addSubscriptionInternal(
        String sessionId, String subsId, String destination, Message<?> message) {

    Expression expression = null;
    MessageHeaders headers = message.getHeaders();
    String selector = SimpMessageHeaderAccessor.getFirstNativeHeader(getSelectorHeaderName(), headers);
    if (selector != null) {
        try {
            expression = this.expressionParser.parseExpression(selector);
            this.selectorHeaderInUse = true;
            if (logger.isTraceEnabled()) {
                logger.trace("Subscription selector: [" + selector + "]");
            }
        }
        catch (Throwable ex) {
            if (logger.isDebugEnabled()) {
                logger.debug("Failed to parse selector: " + selector, ex);
            }
        }
    }
    this.subscriptionRegistry.addSubscription(sessionId, subsId, destination, expression);
    this.destinationCache.updateAfterNewSubscription(destination, sessionId, subsId);
}
```

헤더 정보는 'header' 변수에 저장되고 이때 저장된 값에는 공격자가 삽입한 악성코드도 있다. 이후 클라이언트가 SEND 명령을 요청하면, 클라이언트가 입력한 문자열 앞에 'Hello'를 붙여 서버가 클라이언트로 전송할 메시지를 생성하게 된다.

- **sendMessageToSubscribers() 함수**

메시지를 생성한 후 이를 배포하기 위해 sendMessageToSubscribers() 함수를 호출하게 된다.

경로: org.springframework.messaging.broker -> SimpleBrokerMessageHandler.class

```
protected void sendMessageToSubscribers(@Nullable String destination, Message<?> message) {
    MultiValueMap<String,String> subscriptions = this.subscriptionRegistry.findSubscriptions(message);
    if (!subscriptions.isEmpty() && logger.isDebugEnabled()) {
        logger.debug("Broadcasting to " + subscriptions.size() + " sessions.");
    }
    long now = System.currentTimeMillis();
```

'message'변수에는 payload, session, destination 정보가 저장되어 있고, payload에는 클라이언트로 전송할 메시지 "Hello, client message!"가 있다. 이것은 findSubscription()함수 호출 시 매개 변수로 넘겨준다.

-**findSubscription() 함수**

이 함수에서는 메시지의 에러 검사를 수행하고, 에러가 존재하지 않을 경우 findSubscriptionInternal() 함수로 destination과 메시지를 전달한다.

경로 : org.springframework.messaging.broker -> AbstractSubscriptionRegsitry.class

```
public final MultiValueMap<String, String> findSubscriptions(Message<?> message) {
    MessageHeaders headers = message.getHeaders();

    SimpMessageType type = SimpMessageHeaderAccessor.getMessageType(headers);
    if (!SimpMessageType.MESSAGE.equals(type)) {
        throw new IllegalArgumentException("Unexpected message type: " + type);
    }

    String destination = SimpMessageHeaderAccessor.getDestination(headers);
    if (destination == null) {
        if (logger.isErrorEnabled()) {
            logger.error("No destination in " + message);
        }
        return EMPTY_MAP;
    }

    return findSubscriptionsInternal(destination, message);
}
```

- findSubscriptionInternal() 함수

이 함수에서는 클라이언트의 세션과 ID값을 result라는 MultiValueMap 타입 객체로 생성하고, 이것을 filterSubscription() 함수를 호출하면서 메시지와 함께 매개변수로 넘겨준다.

경로 : org.springframework.messaging.broker -> DefaultSubscriptionRegistry.class

```
protected MultiValueMap<String, String> findSubscriptionsInternal(String destination, Message<?> message) {
    MultiValueMap<String, String> result = this.destinationCache.getSubscriptions(destination, message);
    return filterSubscriptions(result, message);
}
```

- filterSubscription() 함수

이 함수에서는 keyset()을 통해 클라이언트의 sessionID와 subid 값을 획득하고, getSubscriptions()함수를 실행시켜 하위 변수들의 값을 추출한다.

경로 : org.springframework.messaging.broker -> DefaultSubscriptionRegistry.class

```
private MultiValueMap<String, String> filterSubscriptions(
        MultiValueMap<String, String> allMatches, Message<?> message) {

    if (!this.selectorHeaderInUse) {
        return allMatches;
    }
```

하위 값을 추출하던 과정에 클라이언트 정보 ID를 통해 getSelectorExpression()을 획득하는데, 이때 공격자가 삽입했던 악성코드가 추출되어 expression 객체에 저장되게 된다.

```
EvaluationContext context = null;
MultiValueMap<String, String> result = new LinkedMultiValueMap<>(allMatches.size());
for (String sessionId : allMatches.keySet()) {
    for (String subId : allMatches.get(sessionId)) {
        SessionSubscriptionInfo info = this.subscriptionRegistry.getSubscriptions(sessionId);
        if (info == null) {
            continue;
        }
        Subscription sub = info.getSubscription(subId);
        if (sub == null) {
            continue;
        }
        Expression expression = sub.getSelectorExpression();
        if (expression == null) {
            result.add(sessionId, subId);
            continue;
        }
        if (context == null) {
            context = new StandardEvaluationContext(message);
            context.getPropertyAccessors().add(new SimpMessageHeaderPropertyAccessor());
        }
if (context == null) {
    context = new StandardEvaluationContext(message);
    context.getPropertyAccessors().add(new SimpMessageHeaderPropertyAccessor());
}
```

Expression객체에 저장되어 있는 악성코드는 getValue()함수를 통해 추출되어 공격자가 원하는 결과가 실행된다.

```
try {
    if (Boolean.TRUE.equals(expression.getValue(context, Boolean.class))) {
        result.add(sessionId, subId);
    }
}
```

null 값으로 저장되어 있던 context 변수에는 StandardEvaluationContext 인터페이스로 값이 저장되는데 이 인터페이스는 Spring expression language 언어기능 및 구성 옵션의 전체 세트를 제공하게 되어 getValue가 크게 제한을 받지 않고 true 값이 될 수 있도록 해준다.

```
if (context == null) {
    context = new StandardEvaluationContext(message);
    context.getPropertyAccessors().add(new SimpMessageHeaderPropertyAccessor());
}
```

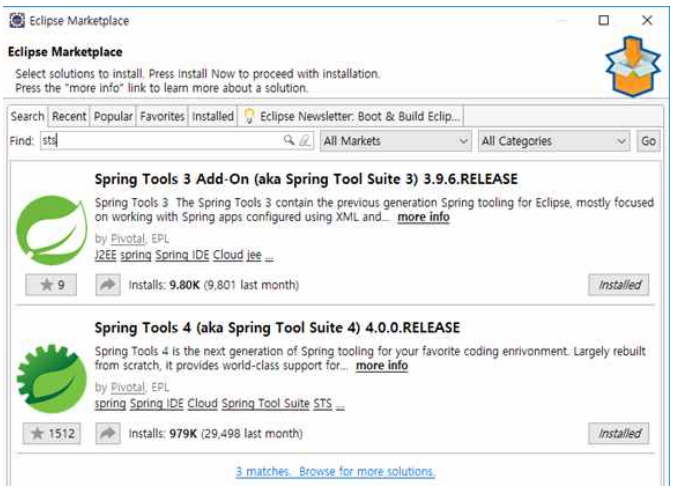
4. 취약점 테스트

4-1. 환경구축

Windows 환경에서 테스트를 진행하였고, 서버를 구축하기 위해 Eclipse에서 spring boot를 설치하여 사용하였다. 설치과정은 다음과 같다.

- Eclipse 설치

  - Eclipse -> Help -> Eclipse Marketplace -> Spring Tools install(STS) 설치



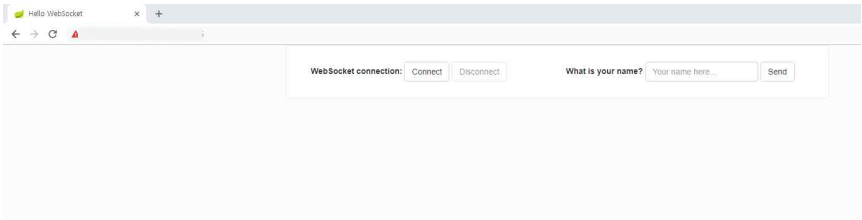
설치가 끝나면 Spring-messaging' 모듈을 활용하여 개발된 메신저 웹 서비스 프로젝트를 다운을 받아 서버에 설치되어있는 Eclipse에 Import한다. 여기에서 사용한 프로젝트는 'gs-messaging-stomp-websocket-2.0.0.RELEASE' 이다.

- 취약한 버전으로는 spring-framework 5.0.4.RELEASE를 사용하였고 '2.0.0.RELEASE'는 .m2\repository\org\springframework\spring-messagomg\5.0.4.RELEASE 안에 위치해 놓는다.

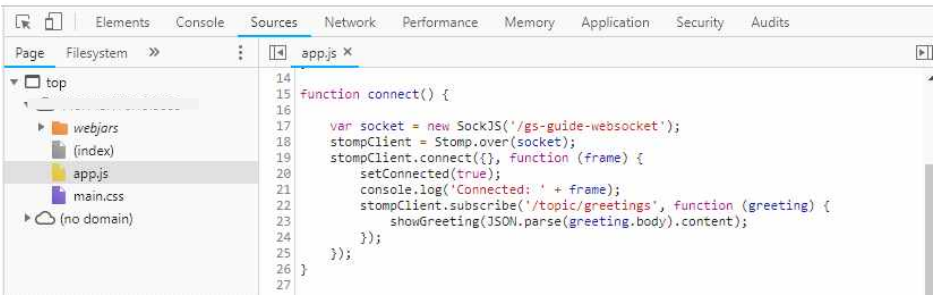
- Eclipse -> File -> Import -> Maven-Existing Maven Projects -> Browse - spring-messaging(소스코드 위치 지정) -> Finish

4-2. 악성코드 삽입

Eclipse에서 웹 서버를 구동시킨 후 크롬에 접속하여 http://서버 주소: 포트 번호/에 접근하면 다음과 같은 웹 사이트를 확인할 수 있다.



웹 사이트에 접속하면 리소스 파일 app.js가 로드되는데 이 파일에는 클라이언트에서 사용할 자바스크립트 함수가 정의되어 있다. 공격자는 이 함수에 악성코드를 추가하여 실행시킴으로써 취약점을 발생시키게 된다.

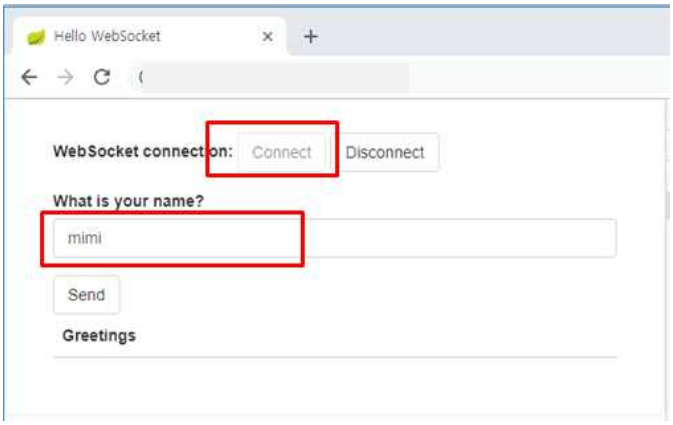


먼저 Consol창을 실행시키고 app.js 함수에서 웹 소켓을 연결하는 connect()함수에 악성코드를 삽입하고 stompClient.subscribe에 코드가 삽입된 변수 'header'를 추가하여 subscribe 명령시 서버에 전송되도록 한다.

성공적으로 전송이 되면 다음과 같은 결과가 나오게 된다.

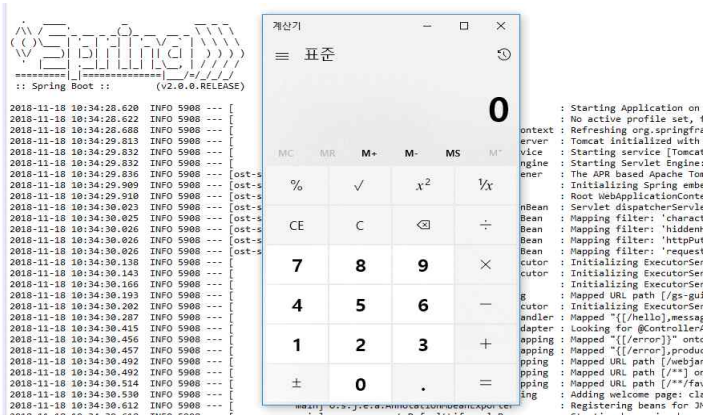
4-3. 웹 소켓 연결 및 문자 전송

이후 connet를 클릭하고 'mimi'라는 문자를 전송한다.



4-4. 원격 명령 실행

결과 서버에서는 공격자가 삽입한 코드인 계산기가 실행되는 것을 확인할 수 있다.



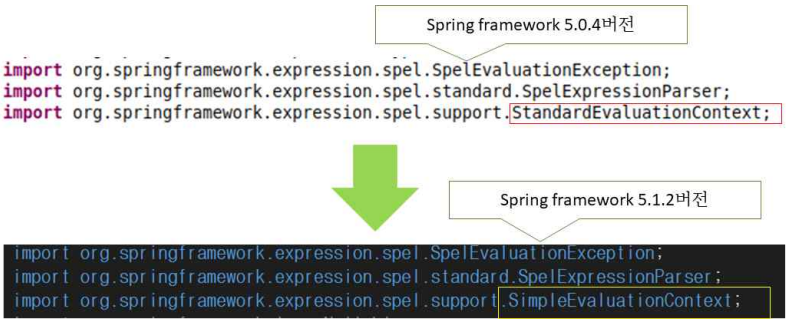
5. 대응방안

1) Spring Framework 업그레이드

스프링(Spring) 공식 홈페이지에서 제공하는 최신 버전으로 업그레이드 한다
Spring Framework 5.0.5 이상
Spring Framework 4.3.16 이상

2) EvaluationContext Interface 전달 방식을 변경

SpEL 언어 기능 및 구성 옵션의 전체 세트를 제공하는 StandardEvaluationContext 대신에 필수적인 SpEL 언어기능 및 구문의 하위 집합만 지원하는 SimpleEvaluationContext 로 대체 한다. 이것은 Java 유형 참조, 생성자 및 bean 참조 등을 제외 시키는 역할을 한다.



이러한 결과 공격자가 보낸 payload를 저장하였다가 실행하던 Expression 객체에서 SimpleEvaluationContext로 인해 실행 제한을 받게 되어 코드가 실행되는 것을 방지할 수 있다.

6. 참고자료

[1] Resarch & Technique] 스프링 프레임워크 메시징 원격코드 실행 취약점
http://blog.naver.com/PostView.nhn?blogId=skinfosec2000&logNo=221282525919&parentCategoryNo=&categoryNo=9&viewDate=&isShowPopularPosts=true&from=search
[2] 제3회 세미나 - Spring 서버 원격코드 시행 취약점(CVE-2018-1270)
https://www.youtube.com/watch?v=qiGyqlJJz9E