

12 : 웹 애플리케이션 보안



학습 목표

- 웹 애플리케이션의 보안에 대해 이해한다.
- 보안 요소 들에 대해 알아본다



1. 서블릿 보안 요소

2. 인증

3. 인가

4. 데이터 비밀보장 및 무결성



보안 >>서블릿 보안의 4 요소

■ 인증 (Authentication)

■ 인가 (Authorization)

■ 비밀보장 (Confidentiality)

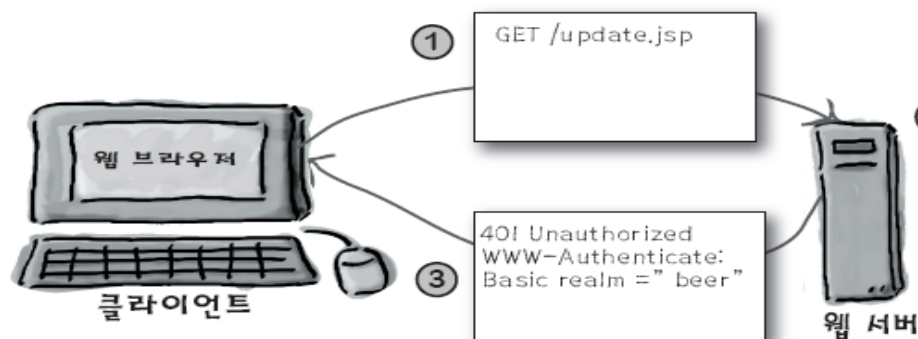
■ 데이터 무결성 (Data Integrity)



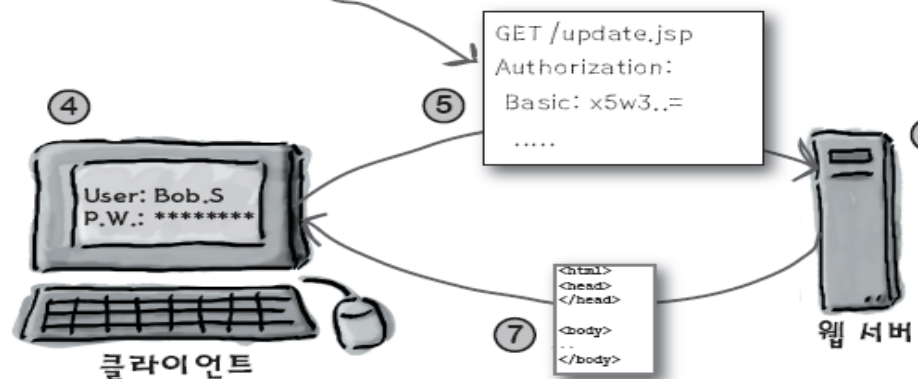
서블릿 보안 >> 인증

■ HTTP 인증

HTTP 관점에서 보면...



인증 내용이 들어 있는 HTTP 헤더



1 브라우저가 제일 먼저 "update.jsp"를 요청합니다.

2 "update.jsp"는 제약 조건이 걸린 자원임을 알아차린 컨테이너는

3 HTTP 401("Unauthorized") 응답을 보냅니다. 여기에는 www-인증 헤더와 보안 영역(realm) 정보가 들어 있습니다.

4 401 코드를 받아본 브라우저, 보안 영역 정보에 기초하여 사용자 이름과 패스워드를 물어봅니다.

5 다시 한번 "update.jsp"를 요청합니다. 하지만 이번 요청엔 보안 관련 HTTP 헤더에 사용자 이름과 패스워드가 들어 있지요.

6 사용자 이름과 패스워드가 일치하는지 체크한 다음, 맞다면, 사용할 수 있는 권한을 인가합니다.

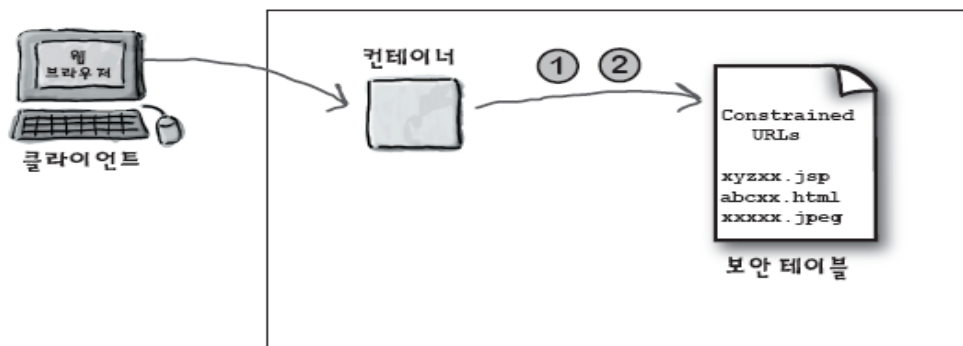
7 보안 관련 체크를 모두 통과했으면, HTML 페이지를 넘겨줍니다. 아니라면 다시 한번 "HTTP 401" 오류를 보내겠죠.



서블릿 보안 >>인증

■ HTTP 인증

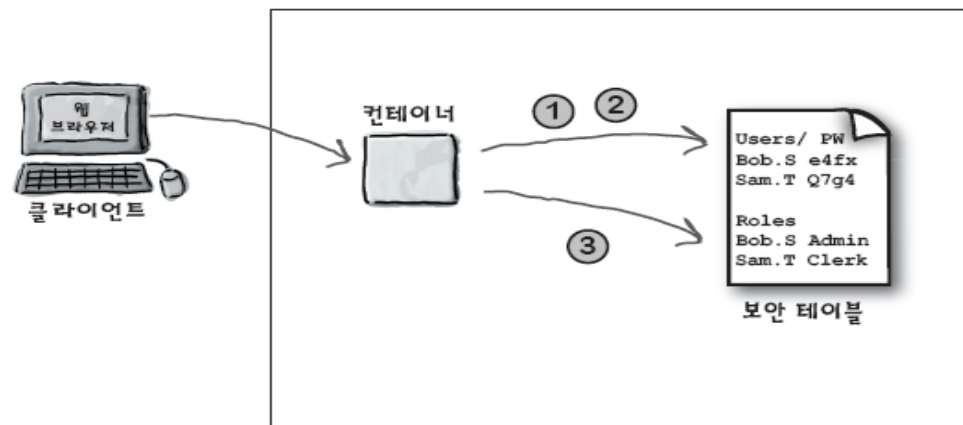
컨테이너 관점에서...



패스워드 없이 날린 첫 번째 요청

1 요청을 접수한 다음, 컨테이너는 이 요청이 "보안 테이블"에 있는 URL인지 체크합니다(컨테이너가 보안 테이블에 어떤 식으로든 보안 정보를 저장하고 있다는 것이겠죠)

2 보안 테이블에 있는 URL이라면, 여기에 무슨 보안 제약이 걸려 있는지 체크하고, 만약 그렇다면 401 코드를 리턴합니다.



패스워드를 넣은 다음 날린 두 번째 요청

1 사용자 이름과 패스워드가 들어 있는 요청을 접수한 컨테이너는 이 요청이 "보안 테이블"에 있는 URL인지 체크합니다

2 보안 테이블에 있는 URL이라면(그리고 보안 제약이 걸려 있다면), 사용자 이름과 패스워드가 일치하는지 체크합니다.

3 사용자 이름과 패스워드가 올바르다면, 사용자가 이 자원을 접근할 수 있는 권한이 있는지(즉 인가되어 있는지) 확인합니다. 만약 그렇다면 클라이언트로 해당 자원을 넘겨줍니다.

서블릿 보안 >>인증

■ 선언적인 인증

4가지 <login-config> 예제

```

<web-app...>
...
  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>
</web-app>

```

— 또는 —

```

<web-app...>
...
  <login-config>
    <auth-method>DIGEST</auth-method>
  </login-config>
</web-app>

```

— 또는 —

```

<web-app...>
...
  <login-config>
    <auth-method>CLIENT-CERT</auth-method>
  </login-config>
</web-app>

```

— 또는 —

```

<web-app...>
...
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/loginPage.html</form-login-page>
      <form-error-page>/loginError.html</form-error-page>
    </form-login-config>
  </login-config>
</web-app>

```

BASIC은 말 그대로 기본이죠. DD에 BASIC이라고 선언만 해두면, 나머지 컨테이너가 알아서 하죠. 즉 제약이 걸린 자원에 대한 요청이 들어오면 알아서 사용자 이름과 패스워드를 질문한다는 말이죠.

현재 가지고 있는 컨테이너가 DIGEST를 지원한다면, 컨테이너가 모든 상세한 처리는 알아서 할 겁니다.

CLIENT 방식 설정도 간단합니다만, 클라이언트에는 반드시 인증서가 있어야 한다는 제약이 있죠. 이 방식은 가장 강력한 보안을 제공합니다.

FORM이 4가지 중 가장 설정하기가 복잡합니다. 다음 페이지에서 좀더 자세히 살펴보도록 하죠.



서블릿 보안 >> 인증

■ 폼 기반 인증

① DD에 정의

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/loginPage.html</form-login-page>
    <form-error-page>/loginError.html</form-error-page>
  </form-login-config>
</login-config>
```

② loginPage.html 일부...

Please login daddy-o

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username">
  <input type="password" name="j_password">
  <input type="submit" value="Enter">
</form>
```

컨테이너와 협업하기 위해 HTML 로그인 폼의 액션
(action)은 j_security_check이어야 함.

HTTP request에 사용자 이름은
j_user_name에 저장되어 있어야 함.

HTTP request에 패스워드는 j_
password에 저장되어 있어야 함.

③ loginError.html 일부...

```
<html><body>
  Sorry dude, wrong password
</body></html>
```



서블릿 보안 >>인증

■ 인증방식 정리

유형	스펙	데이터 무결성	주석
BASIC	HTTP	Base64 - 약함	HTTP 표준. 모든 브라우저가 지원함
DIGEST	HTTP	좀더 강력한 방식 - SSL 만큼은 아님	HTTP, J2EE 컨테이너에서 옵션 사항임
FORM	J2EE	가장 약함, 암호화 안됨	사용자 정의 로그인 화면 지원
CLIENT-CERT	J2EE	강력함 - 공인 키(PKC)	강력하지만 사용자가 인증서를 가지고 있어야 함



서블릿 보안 >>인가

■ 역할(롤) 정의하기

tomcat-users.xml의 <role> 항목

제공자마다 사용자, 역할을 정의하는 방식이 다를 수 있음.

```
<tomcat-users>
  <role rolename="Admin"/>
  <role rolename="Member"/>
  <role rolename="Guest"/>
  <user username="Annie" password="admin" roles="Admin, Member, Guest" />
  <user username="Diane" password="coder" roles="Member, Guest" />
  <user username="Ted" password="newbie" roles="Guest" />
</tomcat-users>
```

톰캣 tomcat-user.xml을 보면 이 구조로 되어 있음. 주의: 한 사용자가 여러 역할을 가질 수 있음.

서블릿 스펙:

web.xml(DD)의 <security-role> 항목

```
<security-role>
  <role-name>Admin</role-name>
  <role-name>Member</role-name>
  <role-name>Guest</role-name>
</security-role>
```

실제 사용 중에 인가 요청을 받으면 컨테이너는 제공자마다 다른 '역할' 정보를 DD <security-role> 항목에 있는 <role-name>과 서로 매핑 합니다.

```
<login-config>
  <auth-method>BASIC</auth-method>
</login-config>
```

인증하려면 <login-config> 항목을 설정해야 한다는 것 잊으면 안 됨!



서블릿 보안 >>인가

■ 자원/메소드 제약 정의하기

DD의 <security-constraint> 항목

```
<web-app...>
...
<security-constraint>
```

```
<web-resource-collection>
```

```
<web-resource-name>UpdateRecipes</web-resource-name>
```

```
<url-pattern>/Beer/AddRecipe/*</url-pattern>
```

```
<url-pattern>/Beer/ReviewRecipe/*</url-pattern>
```

```
<http-method>GET</http-method>
```

```
<http-method>POST</http-method>
```

```
</web-resource-collection>
```

```
<auth-constraint>
```

```
<role-name>Admin</role-name>
```

```
<role-name>Member</role-name>
```

```
</auth-constraint>
```

```
</security-constraint>
```

```
</web-app>
```

틀이 사용하는 이름입니다. 반드시 있어야 하는 항목이죠.
두 번 다시 안나오니 안심하세요.

<url-pattern> 항목에다 제약을 걸 자원을 정의합니다.

<http-method> 항목으로 <url-pattern>에 정의한 자원에 제약(제한)을 걸 HTTP 메소드를 정의합니다.

<auth-constraint> 항목에는 어떤 역할을 가진 사용자들이 어떤 HTTP 메소드(제약이 걸린)를 호출할 수 있는지 정의합니다. 즉 여기에는 정의된 URL 패턴에 대하여 GET, POST를 보낼 수 있는 역할이 무엇인지 코딩되어 있죠.



서블릿 보안 >>인가

■ <web-resource-collection>

- 반드시 있어야 하는 항목
- 주요 하위 항목 : <url-pattern>, <http-method>
- 자원에 대한 요청에 제약을 걸기 위해서는 URL 패턴과 HTTP 메소드가 같이 사용되어야 함.
- <url-pattern> 에 들어가는 값은 서블릿 표준명명 규칙 및 매핑규칙에 따라야 함
- <url-pattern> 항목은 하나 이상이어야 한다
- <http-method> 에 들어가는 값은 GET, POST, PUT, TRACE, DELETE, HEAD, OPTIONS 이다
- <http-method> 를 사용하지 않는 것은 모든 메소드에 제약을 건다는 의미임
- <security-constraint>안에 하나 이상의 <web-resource-collection> 을 포함할 수 있음



서블릿 보안 >>인가

■ <auth-constraint> , <role-name>

— <auth-constraint> 규칙 —

- <security-constraint> 안에 있는 <auth-constraint>는 옵션 항목입니다.
- <auth-constraint> 항목이 있다는 말은, 관련 URL에 대하여 인증을 실시하라 라고 컨테이너에게 지시하는 것입니다.
- <auth-constraint>가 없다면, 해당 URL에 대하여 인증 없이도 접근할 수 있다는 것을 의미합니다.
- 유지보수를 위하여 <auth-constraint> 안에 <description>을 작성할 것을 추천합니다.
















— <role-name> 규칙 —

- <auth-constraint> 안에 있는 <role-name>은 옵션 항목입니다.
- <role-name> 항목이 있다는 말은, 이 역할은 접근을 허용하라 라고 컨테이너에게 지시하는 것입니다.
- <auth-constraint>에 <role-name>이 하나도 없다면, 어떤 사용자도 접근할 수 없다는 말입니다.
- <role-name>*</role-name>으로 정의되어 있다면, 모든 사용자가 다 접근할 수 있다는 말입니다.
- <role-name> 항목 값은 대소문자를 구분합니다.



서블릿 보안 >>인가

■ <auth-constraint> 작동법

<auth-constraint> 컨텐츠	접근할 수 있는 보안 역할	관리자, 회원, 손님 회원, 손님
<pre><security-constraint> <auth-constraint> <role-name>Admin</role-name> <role-name>Member</role-name> </auth-constraint> </security-constraint></pre>	관리자 회원	  
<pre><security-constraint> <auth-constraint> <role-name>Guest</role-name> </auth-constraint> </security-constraint></pre>	손님	  
<pre><security-constraint> <auth-constraint> <role-name>*</role-name> </auth-constraint> </security-constraint></pre>	모든 사람	  
<auth-constraint>가 없는 경우	모든 사람	  
<pre><security-constraint> <auth-constraint/> </security-constraint></pre>	아무도 안 됨	  

이것 두 개는 똑같은
의미입니다.

이코, 공(Empty) 태그를 넣었다면,
어떤 역할도 접근할 수 없다는 말이죠.



서블릿 보안 >>인가

■ <security-constraint> 이 동일한 자원에 대해 중복 정의하는 경우

컨텐츠 ㉠	컨텐츠 ㉡	UpdateRecipes에 접근 가능한 사용자
1 <pre><auth-constraint> <role-name>Guest</role-name> </auth-constraint></pre>	<pre><auth-constraint> <role-name>Admin</role-name> </auth-constraint></pre>	손님과 관리자
2 <pre><auth-constraint> <role-name>Guest</role-name> </auth-constraint></pre>	<pre><auth-constraint> <role-name>*</role-name> </auth-constraint></pre>	모든 접근 가능
3 <pre><auth-constraint/></pre> <p>내용이 없는 공 태그</p>	<pre><auth-constraint> <role-name>Admin</role-name> </auth-constraint></pre>	모두 접근 불가능
4 <auth-constraint> 항목이 없음	<pre><auth-constraint> <role-name>Admin</role-name> </auth-constraint></pre>	모든 접근 가능



서블릿 보안 >>인가

■ `isUserRole()` 메소드

- 사용자 역할에 따라 다르게 구현할 수 있음
- `isUserRole()` 호출하기 전, 사용자는 반드시 인증을 거쳐야 한다. 인증되지 않은 사용자는 이 메소드를 호출하면 `false` 를 리턴
- 컨테이너는 `isUserRole()` 인자로 넘어온 값과 정의된 사용자 역할을 서로 비교
- 사용자가 해당 역할이라면 `true` 를 리턴



서블릿 보안 >>인가

isUserInRole() 메소드 사용 예

서블릿 코드

```
if( request.isUserInRole("Manager")) {  
    // UpdateRecipe 페이지를 호출합니다.  
    ...  
}  
else {  
    // ViewRecipe 페이지를 호출합니다.  
    ...  
}
```

<security-role-ref> 항목이 없다면
"Manager"라는 <security-role>이 없기 때문에
작동하지 않을 겁니다.

배포 서술자

```
<web-app...>  
  <servlet>  
    <security-role-ref>  
      <role-name>Manager</role-name>  
      <role-link>Admin</role-link>  
    </security-role-ref>  
    ...  
  </servlet>  
  ...  
</web-app>  
<web-app...>  
  <security-role>  
    <role-name>Admin</role-name>  
    <role-name>Member</role-name>  
    <role-name>Guest</role-name>  
  </security-role>  
  ...  
</web-app>
```

<security-role-ref>에 프로그램에서
사용한 역할 이름과 선언적인 <security-
role>을 서로 매핑합니다.



조심하세요!

프로그램에서 사용한 역할 이름이 실제 <security-role>에 있어도 컨테이
너는 <security-role-ref>에 있는 것을 먼저 참조합니다

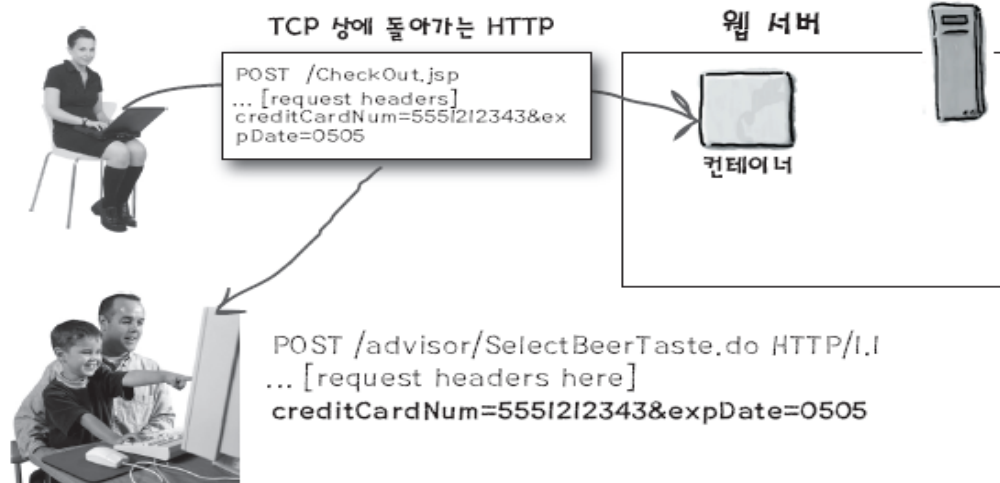
컨테이너는 isUserInRole() 메소드를 만나면, 인자로 들어온 값이 우선 <security-role-ref>에
있는지 찾아봅니다. 있다면, 당연히 이걸 쓰죠. 이걸 이 이름이 <security-role>에 있는 경우에
도 그렇습니다. 예를 들어, "Manager"라는 역할이 실제 있는데, 회사에서는 프로그램에서 생각하
는 의미가 아닌 전혀 다른 의미로 사용하고 있고, 오히려 "Admin"이 의미 상으로 볼 때 가깝다면,
"Manager"를 "Admin"으로 매핑하면 됩니다. 프로그램, 선언 둘 다에 동일한 역할 이름이 있다면,
<security-role-ref>가 우선 적용된다는 것을 기억하세요.



서블릿 보안 >> 비밀 보장과 데이터 무결성

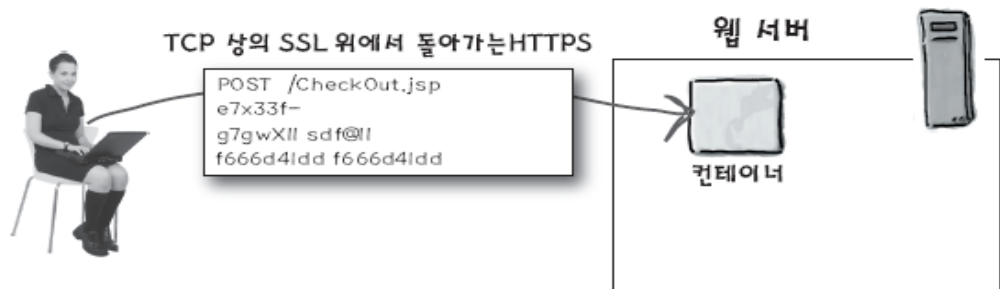
■ 안전한 전송 : HTTPS 이용

HTTP 요청-보안이 안된 방식



나쁜 이즈드롭퍼가 사용자 신용카드 정보가 들어있는 HTTP 요청을 훔쳐보고 있군요. 데이터는 보안 처리되지 않았기에, POST 방식으로 전송되는 데이터 몸체에, 이 정보는 읽을 수 있는 형태로 그대로 나와 있죠. 행복해 하는 저 이즈드롭퍼 얼굴 보입니까?

SSL 기반의 안전한 HTTPS 요청



나쁜 이즈드롭퍼가 사용자 신용카드 정보가 들어 있는 HTTP 요청을 훔쳐보고 있군요.

하지만 이번은 경우가 다르죠. 강력한 SSL 기반 HTTPS로 보안 처리하여 데이터를 전송하기 때문에 데이터를 훔쳐볼 수 없죠.



서블릿 보안 >> 비밀 보장과 데이터 무결성

■ 데이터 기밀성과 데이터 무결성 분리 선언

```
<web-app...>
...
<security-constraint>

    <web-resource-collection>
        <web-resource-name>Recipes</web-resource-name>
        <url-pattern>/Beer/UpdateRecipes/*</url-pattern>
        <http-method>POST</http-method>
    </web-resource-collection>

    <auth-constraint>
        <role-name>Member</role-name>
    </auth-constraint>

    <user-data-constraint>
        <transport-guarantee>CONFIDENTIAL</transport-guarantee>
    </user-data-constraint>

</security-constraint>
</web-app>
```

바로 이것이지요! 데이터 기밀성 및 무결성은 바로 `<user-data-constraint>` 항목에서 모두 선언합니다.

NONE이라고 설정하지 않으셨죠! 데이터를 보호하지 않을 것이면서 `<user-data-constraint>`를 사용하지 않을 테니까요.

가독성을 위하여 위 세 항목을 함께 두세요:

“회원(Member)”만 UpdateRecipes 디렉토리에 있는 자원에 POST 요청을 날릴 수 있습니다. 물론 전송할 데이터를 안전한 방식으로 보호해서 말이죠.

`<transport-guarantee>`에 들어갈 수 있는 값

NONE

디폴트 값. 데이터 보호를 하지 않겠다는 의미죠.

INTEGRAL

전송 중 데이터가 변경되지 않음을 보장한다는 말이죠.

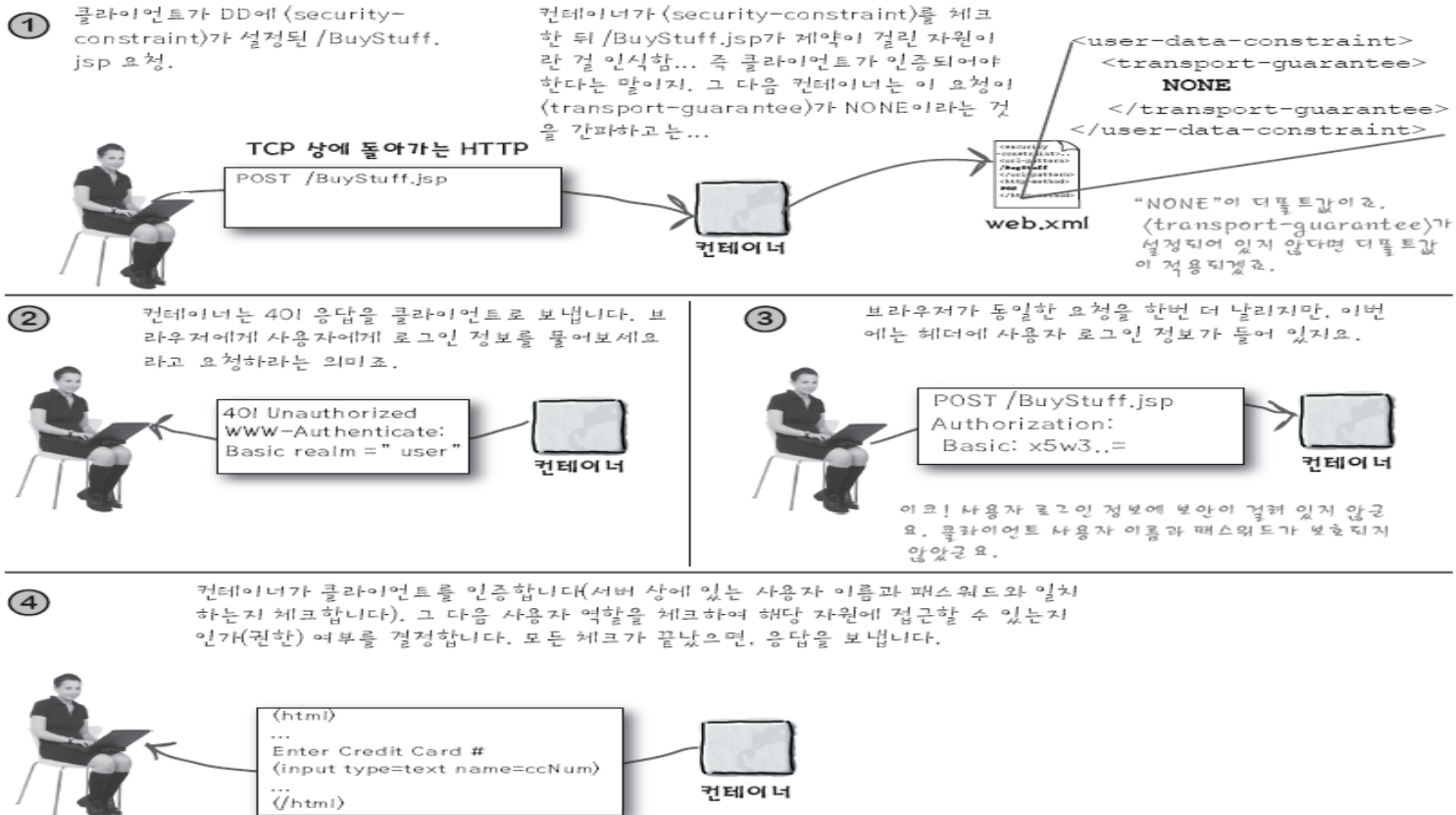
CONFIDENTIAL

전송 중 그 누구도 데이터를 훔쳐보지 않았음을 보장한다는 말이죠.



서블릿 보안 >> 비밀 보장과 데이터 무결성

■ 비인증 클라이언트가 제약에 걸려 있지만 전송 보장 방식이 설정되지 않은 자원에 요청을 보내는 경우 처리





서블릿 보안 >> 비밀 보장과 데이터 무결성

■ 비인증 클라이언트가 제약이 걸려 있으며, 전송 보장 방식이 CONFIDENTIALITY 인 자원에 요청을 보내는 경우 (1)

①

클라이언트가 DD에 <security-constraint>가 설정되어 있으며 전송 보장 방식이 설정된 /BuyStuff.jsp 요청.

컨테이너가 제약이 걸린 자원의 전송을 보장해서 보내야 한다는 사실을 미리 알고 있고, 그 다음 들어온 요청을 체크해보니 전혀 보안이 되지 않은 상태로 날아왔다는 것을 알고는...

```
<user-data-constraint>
  <transport-guarantee>
    CONFIDENTIAL
  </transport-guarantee>
</user-data-constraint>
```

TCP 상에 돌아가는 HTTP

POST /BuyStuff.jsp HTTP



컨테이너

DD

②

컨테이너는 301 응답을 클라이언트로 보냅니다. 즉 안전한 전송 방식으로 요청을 다시 보낼 것을 브라우저에게 요청하는 거죠.

그래, "301"은 일반적인 재전송 때 사용하는 거지. 하지만 여기에는 다음과 같은 내용이 숨어있지. "헤이~ 다시 올 땐 말야 안전한 방식으로 부탁해, 그리고 요청 다시 보내야 한다는 것이지마, 알겠지"

301 Redirect
Location:HTTPS://...

컨테이너





서블릿 보안 >> 비밀 보장과 데이터 무결성

■ 비인증 클라이언트가 제약이 걸려 있으며, 전송 보장 방식이 CONFIDENTIALITY 인 자원에 요청을 보내는 경우 (2)

③

브라우저가 동일한 요청을 한번 더 날리지만, 이번에는 전송 방식을 바꿔 보안 연결을 해서 날리죠. 즉 요청하는 자원은 똑같지만, 프로토콜이 HTTPS로 바뀌지요.



POST /BuyStuff.jsp HTTPS



컨테이너

④

이제 컨테이너는 이 자원에 제약이 걸려있다는 걸 알고는 사용자 인증을 요청합니다. 즉 브라우저에게 “401” 응답을 보내 인증 절차를 밟을 걸 요구합니다.



401 Unauthorized
WWW-Authenticate:
Basic realm = "user"



컨테이너

⑤

브라우저가 동일한 요청을 한번 더 날리지만(예 이것까지 해서 3번째죠), 이번에는 헤더에 사용자 로그인 정보가 들어 있습니다. 물론 이 데이터는 보안 연결을 통해서 날아오겠죠. 앞 페이지와는 달리 사용자 로그인 정보는 안전하게 전송됩니다.



POST /BuyStuff.jsp
Authorization:
Basic: x5w3.,=



컨테이너

정리: 요청을 접수하면, 첫 번째로 컨테이너는 <transport-guarantee> 항목을 체크합니다. 뭔가 설정되어 있다면, 다음과 같이 질문을 던지죠 “이 요청이 보안 연결을 통해 들어온 것인가요?”라고 말이죠. 만약 보안 연결이 아니라면, 컨테이너는 인종/인간 관련 정보는 거들떠 보지도 않고선 다음과 같이 통명스럽게 얘기하죠. “이봐 자네가 안전하다는 걸 보장할 수 있을 때 다시 말을 걸라구...”