Hacking cookies in modern web applications and browsers

Author: Dawid Czagan, Silesia Security Lab
Contact: dczagan@silesiasecuritylab.com

1. Introduction

Since cookies store sensitive data (session ID, CSRF token, etc.) they are interesting from attacker's point of view. As it turns out, quite many web applications (including sensitive ones like bitcoin platforms) have cookie related vulnerabilities that lead for example to user impersonation, remote cookie tampering, XSS and more.

Developers tend to forget that multi-factor authentication will not help, when cookies are insecurely processed. Security evaluators underestimate cookie related problems. Moreover, there are problems with secure processing of cookies in modern browsers and browser dependent exploitation can be used to launch more powerful attacks.

That's why secure cookie processing (from the perspective of web application and browser) seems to be a subject worth discussing.

2. Secure flag

When cookie doesn't have Secure flag set, then it can be sent over insecure HTTP (provided that HSTS is not used; HSTS is described in the next section). When this is a case, the attacker controlling the communication channel between a browser and a server can read this cookie. If the cookie stores session ID, then disclosure of this cookie over insecure HTTP leads to user impersonation.

If cookie has Secure flag set, then it will only be sent over secure HTTPS. Thus the attacker controlling the communication channel can't read it. Although confidentiality is satisfied, there is a problem with integrity. It turns out, that insecure HTTP response can overwrite a cookie with Secure flag. This way insecure HTTP has an impact on secure HTTPS. Modern browsers rely on RFC 6265 (cookie processing is described in this document). That's why this problem occurs in all browsers.

3. HTTP Strict Transport Security (HSTS)

When HSTS is not used, then requests can be sent over insecure HTTP or secure HTTPS. That's why the problem with disclosure of a cookie over insecure HTTP can happen, when cookie doesn't have Secure flag set (described in the previous section).

If HSTS is used, then all requests will be sent over secure HTTPS. Thus cookie with sensitive data (session ID, CSRF token, etc.) will not be disclosed over insecure HTTP (Secure flag is not needed in this scenario). The problem is that HSTS is not supported in all browsers. For example Internet Explorer 10 doesn't support HSTS. That's why it's recommended to use both HSTS and Secure flag.

4. Importance of regeneration

When a cookie with session ID is not regenerated after successful authentication, then attacker can impersonate a user. The attacker learns user's cookie with session ID before user logs in (XSS, disclosure over insecure HTTP, etc.). Then user comes and authenticates. After successful authentication the user still uses the same session ID (it has not been regenerated) and the attacker already knows this value (he learned this value when user was logged out). This way user impersonation via lack of session ID regeneration is possible. Thus it's important to regenerate cookies with sensitive data (session ID, CSRF token, etc.) after successful authentication.

5. Server-side invalidation

When user logs out, then deleting user's cookie with session ID from the browser is not enough. If the cookie has not been invalidated on the server-side, then user impersonation is possible. When this is a case, the user looks like he was logged out, but he is still logged in, as there was no server-side invalidation of his session ID at the time of logging out. To impersonate a user the attacker needs to learn the value of user's session ID before user logs out. There is a number of ways to do it (XSS, disclosure over insecure HTTP, etc.).

6. HttpOnly flag

When a cookie doesn't have HttpOnly flag set, then JavaScript can read a value of this cookie. That's why XSS attack leads to user impersonation if there is no HttpOnly flag set for a cookie with session ID.

When a cookie has HttpOnly flag set, then attacker can't read a value of the cookie in case of XSS attack. The problem is that access permissions are not clearly specified in RFC 6265. It turns out, that cookie with HttpOnly flag can be overwritten in Safari 8.

If user's cookie with session ID is overwritten, then the following attack scenarios can happen:

I. <u>Switching a user to attacker's account</u>
In this case user's session ID is overwritten with attacker's session ID. User thinks that he is using his own account, but enters sensitive information (for example credit card data) to the attacker's account. Thus confidentiality is affected.

II. <u>User impersonation</u>
In this case user's session ID is overwritten with session ID that is not recognized by the web server at the time of overwriting. If session ID is not regenerated after successful authentication, then authenticated user will use session ID that is already known by the attacker. Thus user impersonation is possible.

7. Domain attribute

If a cookie doesn't have specified Domain attribute, then it will only be sent to the domain from which it originated (according to RFC 6265). This way cookie from one domain will not be sent to subdomains of this domain.

The problem is that Internet Explorer 11 will send this cookie also to all subdomains of the domain from which it originated. The following attack scenarios can happen:

I. Cross-origin cookie leakage
Let's assume that sensitive application is hosted on exemplary domain example.com (for example example.com/wallet). This application is sensitive and it was thoroughly analyzed from security point of view. There are also insensitive applications hosted on subdomains of example.com (for example x.example.com and y.example.com) and these applications were not thoroughly analyzed from security point of view. Due to the problem related to Internet Explorer 11 and Domain attribute processing, XSS attack launched on insensitive x.example.com has access to a sensitive cookie that was supposed to be limited only to example.com (it's assumed that this cookie doesn't have HttpOnly flag set).

II. Cookie leakage to externally managed subdomain (shared hosting)
Let's assume that example.com is shared hosting platform and there is a sensitive application hosted on this domain (for example example.com/clients). In this scenario x.example.com and y.example.com are subdomains that are externally managed (not managed by a company offering a shared hosting). Due to the problem described in this section of the paper, these externally managed subdomains have access to a sensitive cookie that was supposed to be limited only to example.com.

8. Cookie tampering

Let's consider a case when user's controlled data (for example the value of GET parameter named lang, which denotes the preferred language) is placed in Set-Cookie header in response. This way the preferred language is stored in a cookie. Dependently on the value of this cookie an appropriate version of the website is displayed (English, German, etc.). Let's assume that attacker can change user's preferred language (via CSRF vulnerability). The impact of this attack is very low. However, it doesn't mean that the attacker is hopeless at this point.

It turns out, that comma separated list of cookies in Set-Cookie header (described in obsoleted RFC 2109) is supported in Safari 8. Thus the attacker can additionally overwrite the value of an arbitrary cookie in a cookie jar of the user. The interesting attack scenario is overwriting a cookie which stores session ID of the user. Then the attacker can switch a user to his account or impersonate a user when session ID is not regenerated after successful authentication. The another attack scenario is launching remotely XSS via cookie (described in the next section). These attack scenarios show that browser dependent exploitation can be used to launch more powerful attacks.

9. Underestimated XSS via cookie

Although different attacks via cookie are possible (for example XSS, SQLi, RCE), XSS via cookie seems to be an underestimated one. The reason is that XSS via cookie is quite often associated with local exploitation, but this is not true. Let's consider different scenarios of launching remotely XSS via cookie:

I. Cross-origin exploitation
Let's assume that XSS via cookie is possible on sensitive y.example.com and there is also XSS vulnerability on insensitive x.example.com. Then remotely exploitable XSS on insensitive x.example.com can be used to set a cookie that will be sent to example.com and all its subdomains (including sensitive y.example.com). Thus XSS on insensitive x.example.com can be used to launch remotely XSS via cookie on sensitive y.example.com.

II. Response splitting
If user's controlled data is placed in Set-Cookie header, then it might be possible to launch response splitting attack. Then carriage return and new line characters are used to split the response and launch XSS. This way, similarly to cross-origin exploitation described above, response splitting on insensitive x.example.cam can be used to launch remotely XSS via cookie on sensitive y.example.com.

III. Cookie tampering
As it was presented in section 8 of this paper, it is possible in Safari 8 to overwrite a value of an arbitrary cookie in a cookie jar of the user via comma separated list of cookies in Set-Cookie header. That's why cookie tampering on insensitive x.example.com can also be used to launch remotely XSS via cookie on sensitive y.example.com.

10. Conclusions

Cookies store sensitive data (session ID, CSRF token, etc.). That's why secure cookie processing is an important subject. It was presented in the paper, that insecure cookie processing is a real problem in modern web applications and browsers. Moreover, there are also problems in RFC 6265 (cookie processing is described in this document and modern browsers rely on it). That's why security engineers/researchers should educate development teams, cooperate with browser vendors and discuss/improve RFC 6265 to make cookie processing more secure.