Web Application Security

Rajendra Kachhwaha
rajendra1983@gmail.com

August 10, 2015

Outline

Missing Function Level Access Control

Most web applications verify function level access rights before making that functionality visible in the UI. However, applications need to perform the same access control checks on the server when each function is accessed. If requests are not verified, attackers will be able to forge requests in order to access functionality without proper authorization.You may be vulnerable if:

1. Does the UI show navigation to unauthorized functions?
2. Are server side authentication or authorization checks missing?
3. Are server side checks done that solely rely on information provided by the attacker?

Using a proxy, browse your application with a privileged role. Then revisit restricted pages using a less privileged role. If the server responses are alike, you're probably vulnerable.

Missing Function Level Access Control

Example Attack Scenarios:

Scenario 1: The attacker simply force browses to target URLs. The following URLs require authentication. Admin rights are also required for access to the admin_getappInfo page.

http : //example.com/app/getappInfo

http : //example.com/app/admin_getappInfo

If an unauthenticated user can access either page, thats a flaw. If an authenticated, non-admin, user is allowed to access the admin_getappInfo page, this is also a flaw, and may lead the attacker to more improperly protected admin pages.

Scenario 2: A page provides an 'action' parameter to specify the function being invoked, and different actions require different roles. If these roles are not enforced, thats a flaw.

Missing Function Level Access Control

How Do I Prevent 'Missing Function Level Access Control'?
The primary recommendations are to establish all of the following:

1. The enforcement mechanism should deny all access by default, requiring explicit grants to specific roles for access to every function.
2. If the function is involved in a work-flow, check to make sure the conditions are in the proper state to allow access.

NOTE: Most web applications dont display links and buttons to unauthorized functions, but this "presentation layer access control" doesnt actually provide protection. You must also implement checks in the controller or business logic.

Cross-Site Request Forgery

A CSRF attack forces a logged-on victims browser to send a forged HTTP request, including the victims session cookie and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victims browser to generate requests the vulnerable application thinks are legitimate requests from the victim.

Cross-Site Request Forgery

Example Attack Scenarios: The application allows a user to submit a state changing request. For ex.:
$http://example.com/app/transferFunds?amount=1500\&dest\_Acct=4673243243$
So, the attacker constructs a request that will transfer money from the victims account to the attackers account, and then embeds this attack in an image request or iframe stored on various sites under the attackers control:
$<imgsrc="http://example.com/app/transferFunds?amount=1500\&dest\_Acct=attackersAcct"width="0"height="0"/>$
If the victim visits any of the attackers sites while already authenticated to example.com, these forged requests will automatically include the users session info, authorizing the attackers request.

Cross-Site Request Forgery

How Do I Prevent 'Cross-Site Request Forgery'?
The primary recommendations are to establish all of the following:

1. The preferred option is to include the unique token in a hidden field.
2. Requiring the user to re-authenticate, or prove they are a user (e.g., via a CAPTCHA) can also protect against CSRF.

Using Components with Known Vulnerabilities

Components, such as libraries, frameworks, and other software modules, almost always run with full privileges. If a vulnerable component is exploited, such an attack can facilitate serious data loss or server takeover.
Applications using components with known vulnerabilities may undermine application defenses and enable a range of possible attacks and impacts.

Using Components with Known Vulnerabilities

How Do I Prevent 'Using Components with Known Vulnerabilities'?
The primary recommendations are to establish all of the following:

1. Identify all components and the versions you are using, including all dependencies.
2. Monitor the security of these components in public databases, project mailing lists, and security mailing lists, and keep them up to date.
3. Establish security policies governing component use, such as requiring certain software development practices, passing security tests, and acceptable licenses.
4. Where appropriate, consider adding security wrappers around components to disable unused functionality and/ or secure weak or vulnerable aspects of the component.

Unvalidated Redirects and Forwards

Web applications frequently redirect and forward users to other pages and websites, and use untrusted data to determine the destination pages.
Without proper validation, attackers can redirect victims to phishing or malware sites, or use forwards to access unauthorized pages.

Unvalidated Redirects and Forwards

How Do I Prevent 'Unvalidated Redirects and Forwards'?
The primary recommendations are to establish all of the following:

1. Dont involve user parameters in calculating the destination.
2. If destination parameters cant be avoided, ensure that the supplied value is valid, and authorized for the user.
   It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server side code translate this mapping to the target URL.

Reference: *https : //www.owasp.org/index.php/*

Blacklist & Whitelist Validation

A **Blacklist** is testing a desired input against a list of negative input's. Basically you would compile a listing of all the negative or bad conditions, then verify that the input received is not one of the bad or negative conditions.

A **Whitelist** is testing a desired input against a list of possible correct input's. To do this you would compile a list of all the good input values/conditions, then verify that the input received is one of this correct conditions.

Blacklists are static in the sense, they prevent 'known bad' from happening. The problem with this is, there are new attack vectors found everyday and you would need to constantly update your black list to be safe. Whitelist on the other hand is more robust because, you can create a filter on exactly what you want.

Which would you think is better?

Blacklist & Whitelist Validation

Never perform validation just on the client side-an attacker can easily bypass these controls. Always validate on the server side.
A Whitelist is the best way to validate input. You will know exactly what is desired and that there is not any bad types accepted. Typically the best way to create a Whitelist is with the use of regular expression's. Using regular expressions is a great way to abstract the Whitelisting, instead of manually listing every possible correct value.
Build a good regular expression. Just because you are using a regular expression does not mean bad input will not be accepted. Make sure you test your regular expression and that invalid input cannot be accepted by your regular expression.
As per my understanding, we have to perform both validations.