



Segurança de aplicativo da web Django

Proteger os dados do usuário é uma parte essencial do design de qualquer site. Explicamos anteriormente algumas das ameaças de segurança mais comuns no artigo [Web security](#) - este artigo fornece uma demonstração prática de como as proteções embutidas do Django lidam com tais ameaças.

Pré-requisitos:

Leia o tópico Programação do lado do servidor " [Segurança do site](#) ". Conclua os tópicos do tutorial do Django até (e incluindo) pelo menos o [Tutorial do Django Parte 9: Trabalhando com formulários](#) .

Objetivo:

Para entender as coisas principais que você precisa fazer (ou não) para proteger seu aplicativo da web Django.

Visão geral

O tópico [Segurança de sites](#) fornece uma visão geral do que significa segurança de sites para o design do lado do servidor e algumas das ameaças mais comuns contra as quais você deve se proteger. Uma das principais mensagens desse artigo é que quase todos os ataques são bem-sucedidos quando o aplicativo da web confia nos dados do navegador.

Importante

A lição mais importante que você pode aprender sobre segurança de sites é **nunca confiar nos dados do navegador** . Isso inclui GET dados de solicitação em parâmetros de URL, POST dados, cabeçalhos HTTP e cookies, arquivos carregados pelo usuário, etc. Sempre verifique e limpe todos os dados de entrada. Sempre presuma o pior.

A boa notícia para os usuários do Django é que muitas das ameaças mais comuns são tratadas pelo framework! O artigo [Security in Django](#) (Django docs) explica os recursos de segurança

do Django e como proteger um site baseado em Django.

Ameaças / proteções comuns

Em vez de duplicar a documentação do Django aqui, neste artigo vamos demonstrar apenas alguns dos recursos de segurança no contexto de nosso tutorial Django [LocalLibrary](#).

Cross site scripting (XSS)

XSS é um termo usado para descrever uma classe de ataques que permite a um invasor injetar scripts do lado do cliente por *meio* do site nos navegadores de outros usuários. Isso geralmente é obtido armazenando scripts maliciosos no banco de dados, onde podem ser recuperados e exibidos para outros usuários, ou fazendo com que os usuários cliquem em um link que fará com que o JavaScript do invasor seja executado pelo navegador do usuário.

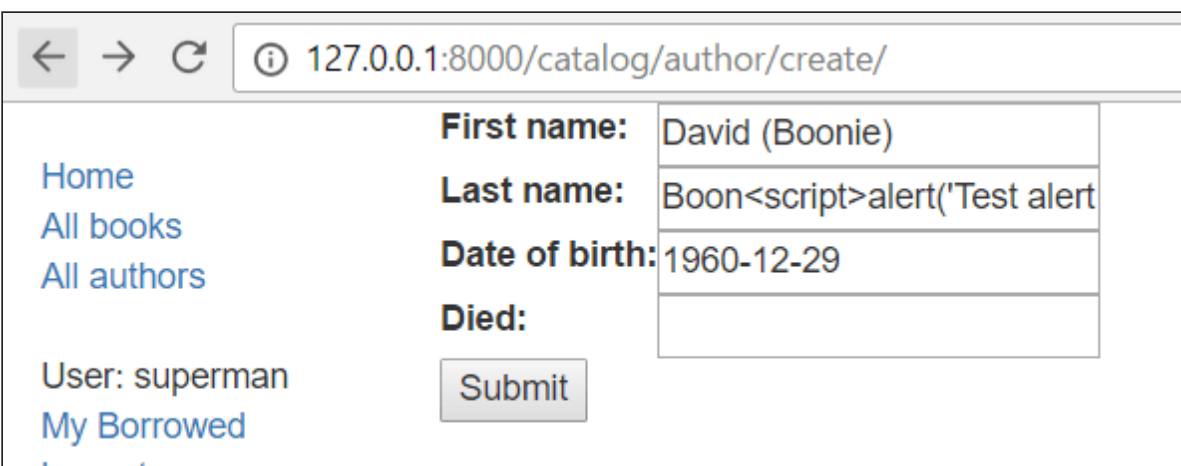
O sistema de template do Django protege você contra a maioria dos ataques XSS [escapando de caracteres específicos](#) que são "perigosos" em HTML. Podemos demonstrar isso tentando injetar algum JavaScript em nosso site LocalLibrary usando o formulário Create-author que configuramos na [Parte 9 do tutorial do Django: Trabalhando com formulários](#).

1. Inicie o site usando o servidor de desenvolvimento (`python3 manage.py runserver`).
2. Abra o site em seu navegador local e faça login em sua conta de superusuário.
3. Navegue até a página de criação do autor (que deve estar no URL:)

<http://127.0.0.1:8000/catalog/author/create/>.

4. Introduzir nomes e detalhes de data para um novo usuário, e em seguida, acrescentar o seguinte texto para o campo Sobrenome:

`<script>alert('Test alert');</script>`.



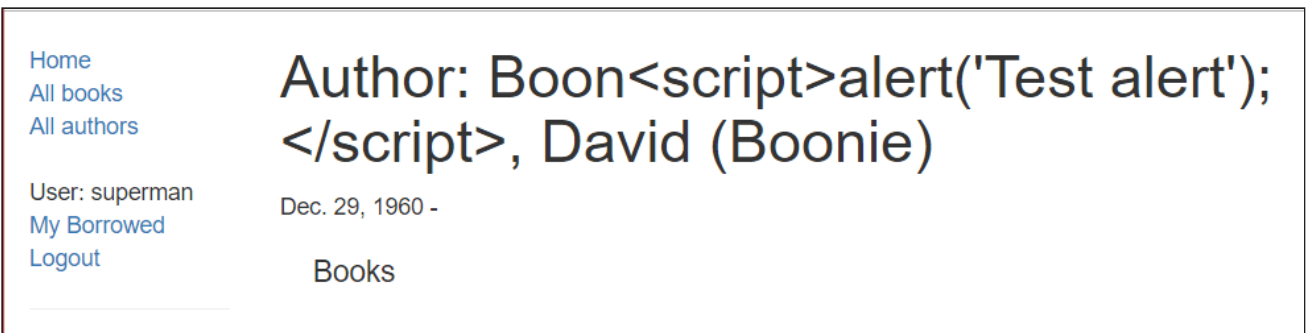
← → ↻ ⓘ	127.0.0.1:8000/catalog/author/create/	
Home All books All authors User: superman My Borrowed Logout	First name:	David (Boonie)
	Last name:	Boon<script>alert('Test alert
	Date of birth:	1960-12-29
	Died:	
	<input type="submit" value="Submit"/>	

[Logout](#)

Observação

Este é um script inofensivo que, se executado, exibirá uma caixa de alerta em seu navegador. Se o alerta for exibido quando você enviar o registro, o site está vulnerável a ameaças XSS.

5. Pressione **Enviar** para salvar o registro.
6. Ao salvar o autor, ele será exibido conforme mostrado abaixo. Por causa das proteções XSS, o `alert()` não deve ser executado. Em vez disso, o script é exibido como texto simples.



If you view the page HTML source code, you can see that the dangerous characters for the script tags have been turned into their harmless escape code equivalents (e.g. `>` is now `>` ;)

```
<h1>Author: Boon<script>alert('Test alert');</script>
```

Using Django templates protects you against the majority of XSS attacks. However it is possible to turn off this protection, and the protection isn't automatically applied to all tags that wouldn't normally be populated by user input (for example, the `help_text` in a form field is usually not user-supplied, so Django doesn't escape those values).

It is also possible for XSS attacks to originate from other untrusted source of data, such as cookies, Web services or uploaded files (whenever the data is not sufficiently sanitized before including in a page). If you're displaying data from these sources, then you may need to add your own sanitisation code.

Cross site request forgery (CSRF) protection

CSRF attacks allow a malicious user to execute actions using the credentials of another user without that user's knowledge or consent. For example consider the case where we have a hacker who wants to create additional authors for our LocalLibrary.

Note

Obviously our hacker isn't in this for the money! A more ambitious hacker could use the same approach on other sites to perform much more harmful tasks (e.g. transfer money to their own accounts, etc.)

In order to do this, they might create an HTML file like the one below, which contains an author-creation form (like the one we used in the previous section) that is submitted as soon as the file is loaded. They would then send the file to all the Librarians and suggest that they open the file (it contains some harmless information, honest!). If the file is opened by any logged in librarian, then the form would be submitted with their credentials and a new author would be created.

```
<html>
<body onload='document.EvilForm.submit() '>

<form action="http://127.0.0.1:8000/catalog/author/create/" method="post"
  <table>
    <tr><th><label for="id_first_name">First name:</label></th><td><input type="text"></td></tr>
    <tr><th><label for="id_last_name">Last name:</label></th><td><input type="text"></td></tr>
    <tr><th><label for="id_date_of_birth">Date of birth:</label></th><td><input type="text"></td></tr>
    <tr><th><label for="id_date_of_death">Died:</label></th><td><input type="text"></td></tr>
  </table>
  <input type="submit" value="Submit" />
</form>

</body>
</html>
```

Run the development web server, and log in with your superuser account. Copy the text above into a file and then open it in the browser. You should get a CSRF error, because Django has protection against this kind of thing!

The way the protection is enabled is that you include the `{% csrf_token %}` template tag in your form definition. This token is then rendered in your HTML as shown below. with a value that

is specific to the user on the current browser.

```
<input type='hidden' name='csrfmiddlewaretoken' value='0QRWHnYVg776y2l66'
```

Django generates a user/browser specific key and will reject forms that do not contain the field, or that contain an incorrect field value for the user/browser.

To use this type of attack the hacker now has to discover and include the CSRF key for the specific target user. They also can't use the "scattergun" approach of sending a malicious file to all librarians and hoping that one of them will open it, since the CSRF key is browser specific.

Django's CSRF protection is turned on by default. You should always use the `{% csrf_token %}` template tag in your forms and use POST for requests that might change or add data to the database.

Other protections

Django also provides other forms of protection (most of which would be hard or not particularly useful to demonstrate):

SQL injection protection

SQL injection vulnerabilities enable malicious users to execute arbitrary SQL code on a database, allowing data to be accessed, modified, or deleted irrespective of the user's permissions. In almost every case you'll be accessing the database using Django's `QuerySets`/models, so the resulting SQL will be properly escaped by the underlying database driver. If you do need to write raw queries or custom SQL then you'll need to explicitly think about preventing SQL injection.

Clickjacking protection

In this attack a malicious user hijacks clicks meant for a visible top level site and routes them to a hidden page beneath. This technique might be used, for example, to display a legitimate bank site but capture the login credentials in an invisible `<iframe>` controlled by the attacker. Django contains [clickjacking protection](#) in the form of the `X-Frame-Options` [middleware](#) which, in a supporting browser, can prevent a site from being rendered inside a frame.

Enforcing SSL/HTTPS

SSL/HTTPS can be enabled on the web server in order to encrypt all traffic between the site and browser, including authentication credentials that would otherwise be sent in plain

text (enabling HTTPS is highly recommended). If HTTPS is enabled then Django provides a number of other protections you can use:

- [SECURE_PROXY_SSL_HEADER](#) can be used to check whether content is secure, even if it is incoming from a non-HTTP proxy.
- [SECURE_SSL_REDIRECT](#) is used to redirect all HTTP requests to HTTPS.
- Use [HTTP Strict Transport Security \(HSTS\)](#). This is an HTTP header that informs a browser that all future connections to a particular site should always use HTTPS. Combined with redirecting HTTP requests to HTTPS, this setting ensures that HTTPS is always used after a successful connection has occurred. HSTS may either be configured with [SECURE_HSTS_SECONDS](#) and [SECURE_HSTS_INCLUDE_SUBDOMAINS](#) or on the Web server.
- Use 'secure' cookies by setting [SESSION_COOKIE_SECURE](#) and [CSRF_COOKIE_SECURE](#) to `True`. This will ensure that cookies are only ever sent over HTTPS.

Host header validation

Use [ALLOWED_HOSTS](#) to only accept requests from trusted hosts.

There are many other protections, and caveats to the usage of the above mechanisms. While we hope that this has given you an overview of what Django offers, you should still read the Django security documentation.

Summary

Django has effective protections against a number of common threats, including XSS and CSRF attacks. In this article we've demonstrated how those particular threats are handled by Django in our *LocalLibrary* website. We've also provided a brief overview of some of the other protections.

This has been a very brief foray into web security. We strongly recommend that you read [Security in Django](#) to gain a deeper understanding.

The next and final step in this module about Django is to complete the [assessment task](#).

See also

- [Security in Django](#) (Django docs)
- [Server side website security](#) (MDN)
- [Securing your site](#) (MDN)

In this module

- [Django introduction](#)
- [Setting up a Django development environment](#)
- [Django Tutorial: The Local Library website](#)
- [Django Tutorial Part 2: Creating a skeleton website](#)
- [Django Tutorial Part 3: Using models](#)
- [Django Tutorial Part 4: Django admin site](#)
- [Django Tutorial Part 5: Creating our home page](#)
- [Django Tutorial Part 6: Generic list and detail views](#)
- [Django Tutorial Part 7: Sessions framework](#)
- [Django Tutorial Part 8: User authentication and permissions](#)
- [Django Tutorial Part 9: Working with forms](#)
- [Django Tutorial Part 10: Testing a Django web application](#)
- [Django Tutorial Part 11: Deploying Django to production](#)
- [Django web application security](#)
- [DIY Django mini blog](#)

Última modificação: 19 de fevereiro de 2021, [por contribuidores MDN](#)

Mude o seu idioma

Inglês (EUA)



Mudar idioma

