Disclosing information with a side-channel in Django

Disclosing information with a side-channel in Django - Dennis Brinkrolf, Sonar.

- Published: 2022-07-26
- Original: <https://www.sonarsource.com/blog/disclosing-information-with-a-side-channel-in-django/>
- Preserved from: https://www.sonarsource.com/blog/disclosing-information-with-a-side-channel-in-django/ (live) on 2026-08-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

TL;DR overview

- Sonar's security research identified a side-channel information disclosure vulnerability in Django, Python's most popular web framework, that allows attackers to infer sensitive information through timing differences in certain comparison operations.
- Timing attacks exploit measurable differences in response time to deduce whether a secret value is correct—a risk when string comparison operations exit early on the first mismatching character.
- Django addressed the vulnerability by using constant-time comparison functions (hmac.compare_digest) in security-sensitive operations; developers implementing custom authentication or token validation in Python should follow the same pattern.
- This research highlights that information disclosure vulnerabilities can exist even in widely trusted frameworks—static analysis combined with security-focused code review is essential for catching these subtle issues.

Django is an open-source Python framework offering modular and reusable components to enable faster development cycles. These components also provide safe foundations for applications, with the core shipping mitigations against most web security mistakes with a strong default configuration.

It's hard, by nature, to get a precise estimate of how many websites rely on a given backend technology stack to operate. Still, its adoption by companies like Mozilla, Instagram, and hobbyist

projects shows how deep it is embedded in the Python ecosystem. At the time of writing, the project has around 65,000 stars on Github.

During research on Django, we undertook to sharpen our static analysis technology, we discovered a way to trick the framework into disclosing sensitive information by interacting with how the data is sorted before displaying it in the interface. Even though this information is obtained through a side-channel based on its relation with other unknown data, we could perform this attack and extract sensitive information in a very reliable manner.

Impact

In cases where users can control how the visualized data will be sorted before display, attackers can leverage this difference to disclose security-sensitive information, like email addresses and password hashes. The basis for this vulnerability is the insecure variable resolution logic in the `dictsort` filter of Django templates. In addition to the leaking of security-related information, we could also demonstrate how this vulnerability could lead to the invocation of an arbitrary method, but with solid limitations.

We responsibly disclosed this finding to the Django maintainers, which prompted them to release fixes on the three supported branches (2.2.26, 3.2.11, and 4.0.1). This vulnerability was later assigned CVE-2021-45116 with a CVSS score of 7.5 (High).

We recommend upgrading applications relying on vulnerable versions of the Django framework to address this risk.

Technical Details

In this section, we first explain how the template engine operates to go into more detail about the cause of insecure variable resolution logic in the `dictsort` filter of Django templates. Afterward, we demonstrate how an attacker can use this limited vector to extract password hashes of all Django users.

Playing with the Django Templating Language

Most frameworks adhering to the MVC (Model, View, Controller) architecture offer ways to programmatically express what the user will be seeing (the "view"). The component in charge of this task is called a *templating engine*; each comes with its own simple language and set of built-in functions (also called filters).

Django supports multiple templating engines, but we'll focus on DTL (for Django Templating Language) in the next sections as this is the default one.

Let's say that you would like to create a page showing every registered user of your database while leaving the ability to your users to change the order in which they will be displayed based on criteria of their choice. For instance, they could want only the most recently updated ones, and later the ones starting with the letter a.

The code below is what most developers would write:

views.py

Copy to clipboard

```
1 from django.contrib.auth import get_user_model
2 from django.shortcuts import render
3
4 def list_users(request):
5    sort = request.GET['sort']
6    user_model = get_user_model()
7    all_users = list(user_model.objects.all())
8
9    to_sort = []
10    for user_obj in all_users:
11          to_sort.append({'users': user_obj})
12
13    context = {'users': to_sort, 'sort': sort}
14    return render(request, 'users.html', context)
```

As we can see on line 7, all users are fetched from the database before being placed in a dictionary with a `sort` attribute on line 13. This object is then passed as context to the template to be rendered:

templates/users.html

Copy to clipboard

```
1 <html>
2 <h1>List all users</h1>
3 {% for e in users|dictsort:sort %}
4    <li> user: {{ e.user.username }}
5 {% endfor %}
6 </html>
```

Notice the use of the built-in filter `dictsort` on line 3, provided with our database entries and the sort criteria defined by the client. This filter will do all the hard work and perform the sort operation for us.

This code is correct and will have the expected behavior; however, it introduces a subtle vulnerability in the application when deployed with a vulnerable release of Django.

What's inside dictsort?

In this section, we deep dive into the implementation of the `dictsort` filter, part of the Django core.

Its code is fairly concise as it relies on the built-in Python function `sorted` the custom function `_property_resolver` to decide the order of the list's elements in the parameter `key`, on line 514:

django/template/defaultfilters.py

Copy to clipboard

```
481 def _property_resolver(arg):
499     try:
500         float(arg)
501     except ValueError:
502         return Variable(arg).resolve
    [...]
507 @register.filter(is_safe=False)
508 def dictsort(value, arg):
513     try:
514         return sorted(value, key=_property_resolver(arg))
515     except (TypeError, VariableDoesNotExist):
516         return ''
```

This custom function first tries to cast the user-controlled argument to a `float` and then instantiates a new `Variable` object if the cast failed. The instantiation of the `Variable` object and the invocation of the `resolve` method is the general logic for resolving template variables in Django.

Given the value of the parameter `arg`, the `Variable` class ensures that it does not try to reference a private method or attribute on line 786—such variables all start with an underscore by convention:

django/template/base.py

Copy to clipboard

```
481 def _property_resolver(arg):
499     try:
500         float(arg)
501     except ValueError:
502         return Variable(arg).resolve
    [...]
507 @register.filter(is_safe=False)
508 def dictsort(value, arg):
513     try:
514         return sorted(value, key=_property_resolver(arg))
515     except (TypeError, VariableDoesNotExist):
516         return ''
```

This custom function first tries to cast the user-controlled argument to a `float` and then instantiates a new `Variable` object if the cast failed. The instantiation of the `Variable` object and the invocation of the `resolve` method is the general logic for resolving template variables in Django.

Given the value of the parameter `arg`, the `Variable` class ensures that it does not try to reference a private method or attribute on line 786—such variables all start with an underscore by convention:

django/template/base.py

Copy to clipboard

```
727 class Variable:
746     def __init__(self, var):
786         if var.find(VARIABLE_ATTRIBUTE_SEPARATOR + '_') > -1 or var[0] == '_':
787             raise TemplateSyntaxError()
790         self.lookups = tuple(var.split(VARIABLE_ATTRIBUTE_SEPARATOR))
```

If such a prefix is present, the code raises an exception to prevent further processing of the reference to the variable. This mechanism effectively prevents attackers from getting to sensitive internal variables via Python builtins.

Values passing the check are later resolved with a method named `_resolve_lookup` to find a variable whose name is contained in `arg`. The following listing shows the interesting parts of `_resolve_lookup`; it was streamlined and slightly simplified to fit this article.

The variable resolution syntax is not limited to accessing attributes and tries four different lookups:

- Line 829: dictionary lookup, e.g., foo.bar to do `foo[bar]` in Python
- Line 837: Attribute lookup, e.g., foo.bar to do `foo.bar` in Python
- Line 843: List-index lookup, e.g., foo.1 to do `foo[1]` in Python
- Lines 851 to 858: Method call or object instantiation without arguments, e.g., foo.bar to do `foo.bar()` in Python

django/template/base.py

Copy to clipboard

```
727 class Variable:
816     def _resolve_lookup(self, context):
825         current = context
826         try:
827             for bit in self.lookups:
828                 try:
829                     current = current[bit]
832                 except:
833                     try:
837                         current = getattr(current, bit)
838                     except:
842                         try:
843                             current = current[int(bit)]
844                         except:
848                             raise Exception
851             if callable(current):
854                 if getattr(current, 'alters_data', False):
855                     raise Exception
856                 else:
857                     try:
858                         current = current()
881         return current
```

The acute reader will have identified a very specific condition on line 854: This is part of the Django templating API and documented on their website. It prevents templating functions from modifying, e.g., in this case, `foo.bar.delete()` unless this `alters_data` attribute is set first. This works as required by the conventions of the MVC architecture, where the "view" plane should not alter the data.

As we can see, Django has done quite a lot to keep the rendering process secure and to disarm the exploitation of untrusted resolution of variables. Despite this surprising primitive that allows calling arbitrary Python methods, the lack of controlled arguments allows us to perform actions such as deleting application files, emptying the database, or modifying the runtime configuration of Django.

However, we want to demonstrate another exploit technique that is little-known.

Disclosing information with a sorting oracle

In this section, we demonstrate an alternative approach to leak security-sensitive information like passwords hashes efficiently and thanks to `dictsort`.

The following example shows how an attacker can extract information from a user object by abusing a sorting oracle:

Copy to clipboard

```
1 user1_obj.username = "sonarsource"
2 user1_obj.password = "scary"
3 user2_obj.username = "admin"
4 user2_obj.password = "admin"
5 value = [ {"user":user1_obj}, {"user":user2.obj} ]

# output of dictsort sorted by the first character of the password
7 [ {"user":user2_obj}, {"user":user1.obj} ] -> 'admin', 'sonarsource'

# output of dictsort sorted by the second character of the password
9 [ {"user":user1.obj}, {"user":user2_obj}] -> 'sonarsource', 'admin'
```

The first 5 lines construct the users to be sorted. Line 7 shows the sorted list after sorting all users by their passwords' first character `(0)`. We see that the user `admin` appears before the user `sonarsource` in the sorted list. Now we sort all users by the second character `(1)` of their passwords. In line 9, we now see that the user `sonarsource` appears before `admin` in the sorted list.

Thus, an attacker could learn something about the individual passwords of the users from the resulting sorting.

However, the attacker only knows that an ASCII character is greater or smaller than another:

Copy to clipboard

```
user2_obj.password.0 (a) < user1_obj.password.0 (s)
user1_obj.password.1 (c) < user2_obj.password.1 (d)
```

For unique identification of each character of the password, every possible ASCII character must appear at every position of the password. In ASCII, there are 128 characters. If each character occurred only once at each position of the password, the attacker would need 128 users to extract the passwords without a wrong result.

In Django, passwords are hashes with an unknown secret and an unknown random salt. Furthermore, the attacker does not know his own password's hashed version. Therefore, changing his password to influence the sorting and learning about the other hashed passwords is not useful. In addition, passwords are not perfectly evenly distributed, and multiple occurrences of the same characters must be expected. The next section demonstrates how an attacker can overcome these difficulties to extract all passwords without errors.

Applying this method to simple hashes

We now have a theoretical attack to leak information using the sorting oracle, and we can apply it to password hashes of registered Django users. To simplify the explanation, we've crafted a small example in the table below. We assume ten users in the database, and a password hash with the format `p[abcd]{2}$`: every user's hash always starts with `p` followed by two characters from the alphabet {a,b,c,d}.

The following table shows in the first column all usernames that are displayed unsorted, and between parentheses is a numerical identifier assigned by an attacker in ascending order. The second column shows the complete password hash of each user that an attacker would like to extract. Remember that the password hash field is not displayed on the interface.

But how does an attacker get the unsorted users in the first column? This output is obtained by sorting with the criteria `user.password.0`. Since the first character of a hash is always a `p` and thus the same for all users, the order of the users remains unsorted since there is no difference between them. We'll call it "unsorted" from now, and with this simple but effective trick of numbering the users, we have created a **primitive** we will need later.

Request 1: unsorted list of users by sorting on the first character of the password

The second table shows the attacker's second request. The first column shows all users sorted by the second character of the hash via the payload `user.password.1`. Keep in mind that the attacker only sees each user's username. However, since an attacker has given each username a unique identifier in the first request, each user can be reassigned to his id. Between parentheses is the second character of the hash of each user that the attacker wants to extract in this request. The second column contains the extracted character hash for each user.

Request 2: sorting based on the second character of the password

But how can the attacker extract the correct character of the hash for each user from the second column? When we constructed the example, we assumed that every character of the hash occurs at every position. In this case, the first User 9 specifies the beginning of group `a`, while the last User 2 defines the end of group `d`. But how can an attacker now organize the remaining users? User 10 could now be in group `a`, or in the next group `b`. To overcome this inaccuracy, we use a simple trick to organize all remaining users into groups (our exploit primitive from the first request). If users are in the same group, the order of the users remains the same even after sorting. If this is not the case, the current user defines the beginning of the next and the end of the last group.

For example, in the first column of the first table, which contains the unsorted users, is User 10 after User 9. However, after sorting by the second character of the hash referring to the first column of the second table, User 10 is still after User 9. Therefore the user with id 10 belongs to group a. However, the next User 6 is after User 10 and this should not happen if User 6 had the same second character as User 10. In this case, the sorting has **rearranged** the order, indicating that another character occurred, so the attacker opens a new group b.

Here it becomes obvious why the unsorted list trick is so effective: the unsorted output of the users can be used to track the users even after their sorting and allows an attacker to precisely define which group the extracted character of each user belongs to.

The last table shows an attacker's third and last request and has the same structure as the previous table. The attacker sorts all hashes by the third character via the payload user.password.2 and can categorize each user into the corresponding groups as before.

Request 3: sorting based on the third character of the password

Finally, the attacker only has to go through all groups for all characters and has extracted the complete hash of each user's password. An interesting fact is that it takes an attacker only three

requests to extract all password hashes of ten users. Each request provided information about all hashes at the same time. Thus, the complexity of the extraction process does not depend on the number of users but is linear to the length of the extracted string. To extract a string of length n, an attacker only needs n+1 requests. The plus one is the first initial request to get an unsorted order (primitive) but can be ignored in case of complexity analysis.

Applying this attack to Django hashes

Let's dive deeper into the structure of password hashes in Django to apply this attack on a real instance.

By default, Django uses the `pbkdf2_sha256` algorithm with `320,000` iterations, a `secret`, and a random `salt` for every user and always starts by default with the string `pbkdf2_sha256`. It should be clear now why the hash in the example above always starts with `p`.

Here is an example of what it looks like:

Copy to clipboard

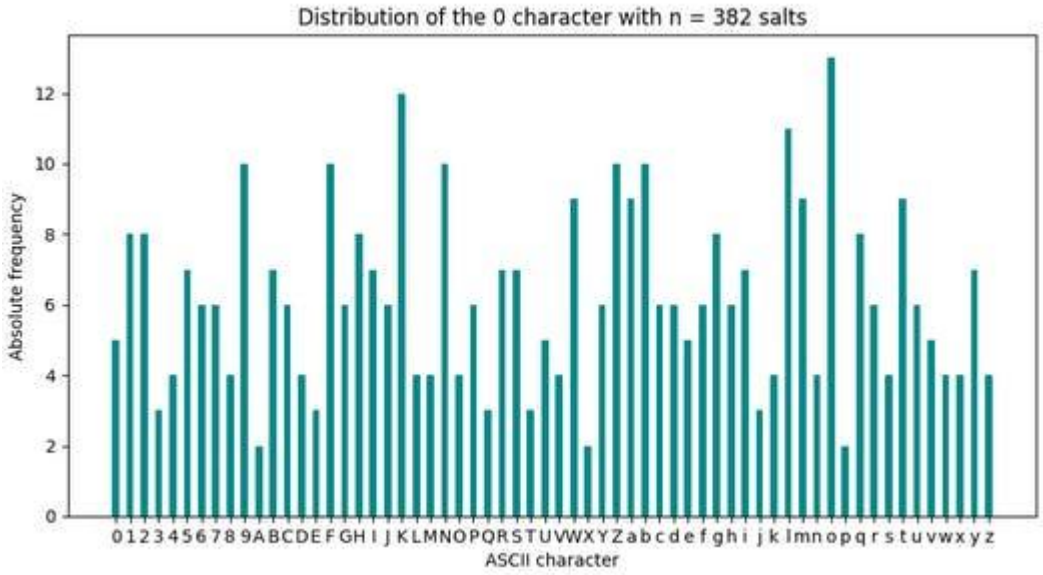
```
pbkdf2_sha256$320000$8ox2uTDNLbz0PZdmLJHoHw$V57Ajo9at9lYiy5C9viva9n0sCMA8JxG5SG1fvO/xMA=
```

The salt `8ox2uTDNLbz0PZdmLJHoHw` is safely generated and always has a length of 22 characters within the charset `[a-zA-Z0-9]`.

The hash, `V57Ajo9at9lYiy5C9viva9n0sCMA8JxG5SG1fvO/xMA=`, is base64-encoded and always has a length of 44 characters within the charset `[a-zA-Z0-9+\/=]` because of the encoding.

The character set of salt is `[a-zA-Z0-9]`, meaning that a maximum of 62 characters can appear in a salt resulting in 62 different groups. To hit each group once, we would need at least 62 users. However, after some experiments, it turns out that, on average, 374 users are necessary for each ASCII character of the salt to occur at least once at each position. Otherwise, an attacker could not assemble all the groups to extract the hashes.

The following figure shows the absolute frequency of each character from the salt charset `[a-zA-z0-9]` for the first position `(0)` of 382 generated salts. For this experiment run, we required 382 generated salts to meet the minimum requirement for an attack, requiring more salts than on average. However, we see that some characters appear more frequently than others which causes multiple occurrences of the same characters. For this reason, we need our primitive unsorted users again, as in the example mentioned above.



As mentioned above, we are lucky that every password hash always starts by default with `pbkdf2_sha256`. If we now sort all users by the first characters of the password hash, a `p` is returned for each user, and here is our "unsorted" primitive again! The algorithm to extract all hashes (Salt + base64_encode(HASH)) is the same as described in the minimal example above.

On average, an attacker needs at least 800 registered users to extract all hashes of all users in only 67 (22 + 44 + 1) requests without a wrong hash.

The exploit would be possible with a smaller number of users but would result in multiple characters being possible for each password hash. There are probably some statistical tricks to reduce the errors, and in the worst case, some hash characters could be guessed by brute force. In the real world, an attacker can wait until the number of users is reached or register new users themselves if possible.

Patch

One way to prevent this oracle sorting vulnerability would be to add an allowlist parameter to the `dictsort` filter, restricting access to fields that the developer didn't explicitly intend, such as password hashes. This is the solution we initially suggested to the maintainers, with the non-negligible impact of breaking backward compatibility.

The maintainers chose to limit the functionalities of dictsort's `_property_resolver` to allow only dictionary and attribute lookups. As a result, an attacker can't call methods or instantiate objects without parameters, nor sort by individual characters of a string.

You can find the official advisory on Django's website and the patch on GitHub.

Timeline

Summary

In this post, we covered the technical details behind a vulnerable variable resolution logic in the `dictsort` filter of Django and showed how an attacker could exploit it to extract sensitive data.

We hope that we will succeed in raising the attention of developers to this little-known vulnerability so that they understand the most critical aspects. We also wanted to demonstrate the capabilities of an attacker and how they can exploit side channels such as subtle differences in output, no matter how small.

We want to thank the maintainers of Django for their fast replies, patches, and very efficient vulnerability disclosure process.
