greenhats®
the one-stop solution for proactive cybersecurity

Report penetration test

Dependev SRL, Marin Gigi Adrian

Meta information

Attribut	Wert
Release	(Internal) PT-1, GF | (External) **Public**
Protection	(Access) MFA | (Encryption) GPG
Scope	**https://main.avodemo.com (Avo v3.0.0.pre12)**
Created on	1.1, 05/29/2023

Project participants

Name (Alias)	Project function	Certification level
Paul Werther (FLX)	Lead Auditor	OSCE³,OSCP,OSED,OSEP,OSWP,OSWE
Anton Strilez (Mys7ic)	Quality Control	OSWE, OSCP, OSEP
Florian Knospe (EffpehKah)	Auditor	OSDA

Table of contents

Executive summary

high level

We conducted a two-day penetration test on the product "Avo", which is a Ruby / Ruby on Rails gem for building administrative interfaces. Since greenhats uses this software for some production environments, it is enforced by internal policy to perform a small pentest / whitebox code audit to identify obvious potential risks. This approach differs from a comprehensive penetration test, which for this type and size of application should be scheduled for at least two weeks.

Although such reports contain sensitive data and most of the time will be for private access only, this report as well as the pentest itself will be part of the public community contribution from greenhats to the incredible Avo project, so it will be publicly accessible. Due to the critical impact on the security of Avo users, we will release any findings after we have confirmed that they have been fixed in all of Avo's latest release channels and all users are notified that they must update to be secure.

The entire code base of the product in version 3.0.0.pre12 was audited using both automated and manual techniques. In addition, the authentication and authorization parts were manually audited. Ruby on Rails itself contains many best practices for security implementations and greatly reduces the risk and potential of common vulnerabilities. In particular, the use of standard active record functions and an external, well-known gem for all search operations makes it more difficult to find typical vulnerabilities.

We discovered a bug in the displaying functionality of html-based content that could lead to hijacking of visitors' or other administrators' browsers. We also discovered a potentially critical security issue in the handling of polymorphic resources. Due to time constraints, we do not provide full exploitation of any of these findings. We believe that applying security mitigations even for potential vulnerabilities should be the main goal of any white box code audit, rather than spending too much time developing a full exploit. While this may be helpful in considering the risk of the specific findings, it is not the main objective of this particular test.

We hope that this small contribution will help to use Avo in a safer way and motivate more security researchers to take a closer look at the application. With that said, let's get into some more technical details.

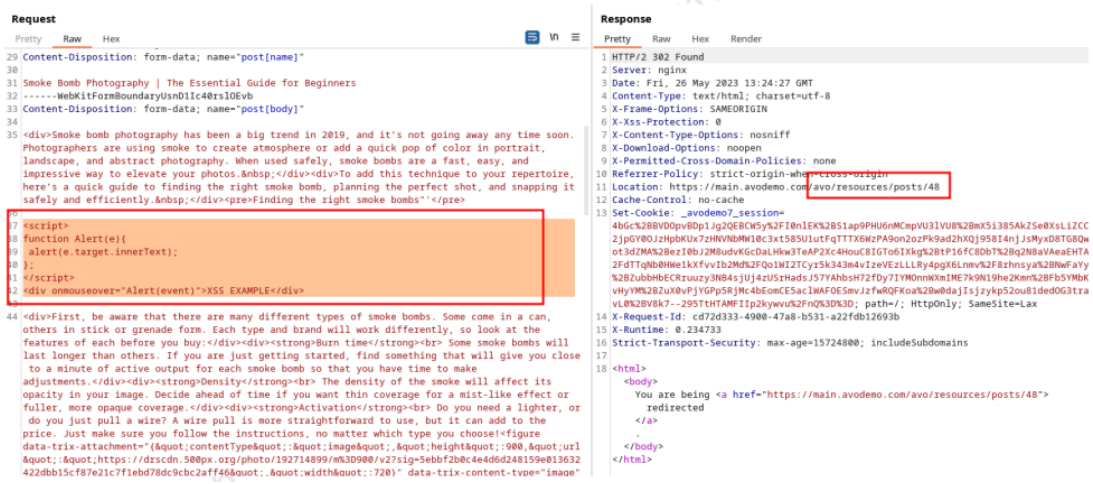
Stored XSS (Cross Site Scripting) (CVE-2023-34103)

critical

Description

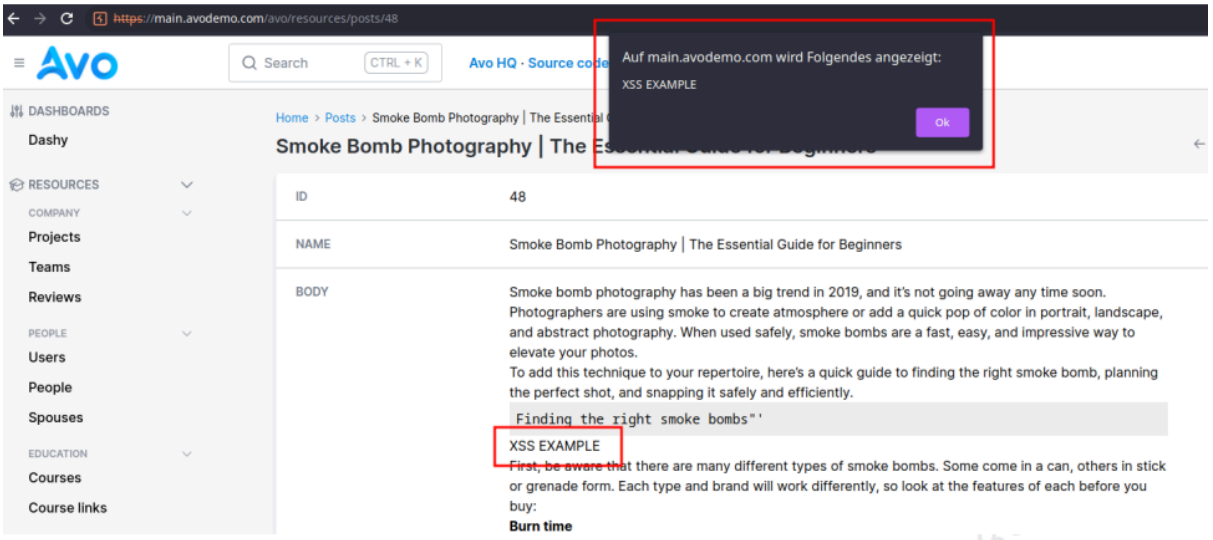
During the analysis of the web application, a rendered field was discovered that did not filter JS / HTML tags in a safe way and can be abused to execute js code on a client side. The trix field[1] uses the trix editor[2] in the backend to edit rich text data which basically operates with html tags. To display the stored data in a rendered view, the HasHTMLAttributes concern is used. This can be exploited by an attacker to store javascript code in any trix field by intercepting the request and modifying the post data, as the trix editor does not allow adding custom html or js tags on the frontend.

Proof of exploitation



Adding javascript in the post request which is used when editing a "post" resource (body is declared as a trix field)

[1] https://docs.avohq.io/3.0/fields/trix.html
[2] https://trix-editor.org/



Successful execution of JS code on live demo environment

Rating

Unlike non-persistent XSS, persistent XSS does not require a social engineering phase. Victims of this attack do not need to be tricked into clicking a link or something like that. However, by exploiting such a vulnerability on this particular target, attackers may be able to gain access to accounts that require special protection, such as administrators of the web service, which is what Avo is primarily intended to be used for.

Recommendation

The content of a field that contains html code should be sanitized using the according rails helper which uses a whitelist of known-safe tags and attributes. Also this security consideration should be applied to the "as_html"[3] attribute as well because it may contain user controlled input as well.

https://api.rubyonrails.org/classes/ActionView/Helpers/SanitizeHelper.html

[3] https://docs.avohq.io/3.0/fields/text.html#as_html

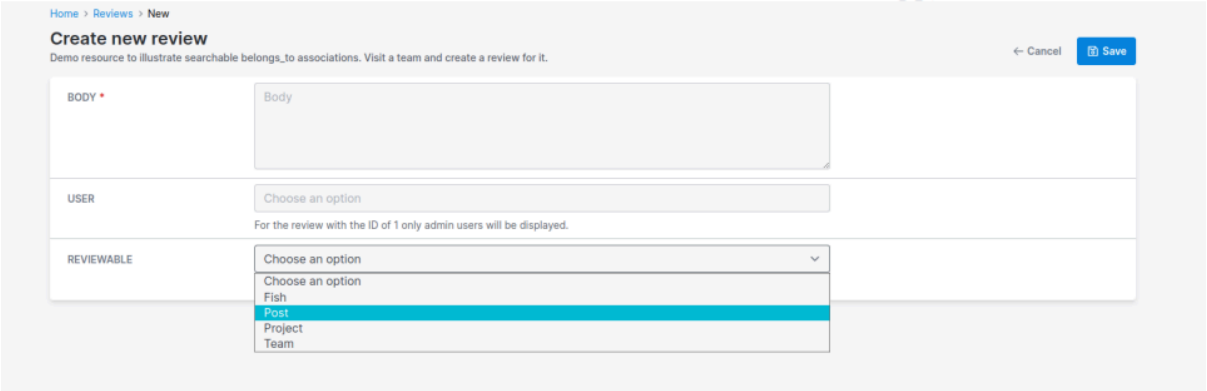
Possible unsafe reflection / partial DoS (CVE-2023-34102)

critical

Description

After reviewing the polymorphic field[4] implementation and performing some black box approaches, we identified a potential security issue related to the use of safe_constantize / constantize[5]. This Rails functionality is capable of searching for classes within the Rails context and returning the class for further use. Because Avo does not validate user input when updating or creating a new polymorphic resource, it is possible to create database entries with completely different or invalid class names than the preselected ones. Avo assumes that the class specified by the user request is a valid one and attempts to work with it, which may result in dangerous behavior and code execution.

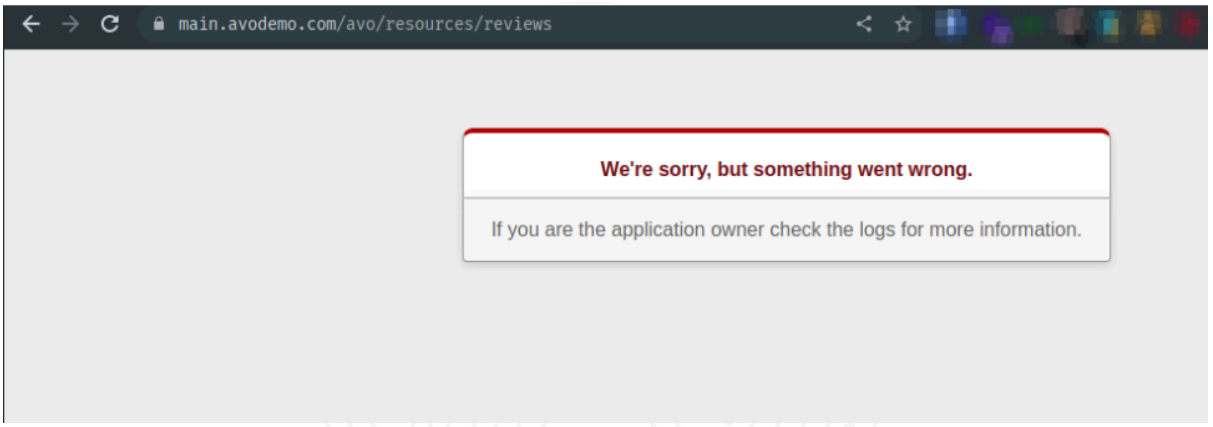
Proof of exploitation



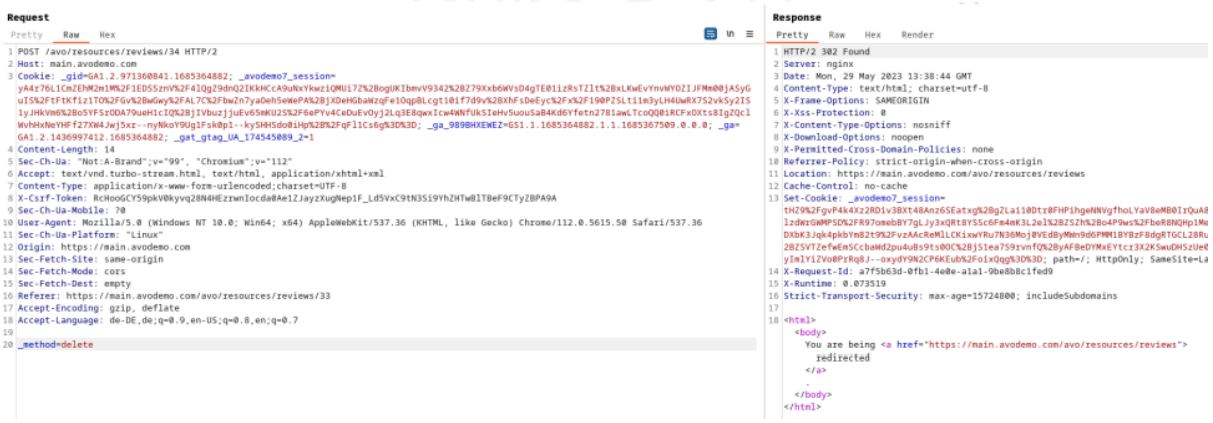
In the test scenario we choose the demo app and the review resource which has a polymorphic reviewable field.

[4] https://docs.avohq.io/2.0/associations/belongs_to.html#polymorphic-belongs-to
[5] https://apidock.com/rails/v5.2.3/ActiveSupport/Inflector/constantize

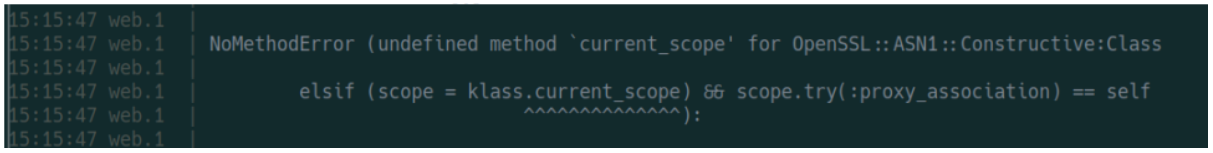
Intercepting the request and switching the review[reviewable_type] from “Fish” to “File” which is a real class inside Rails



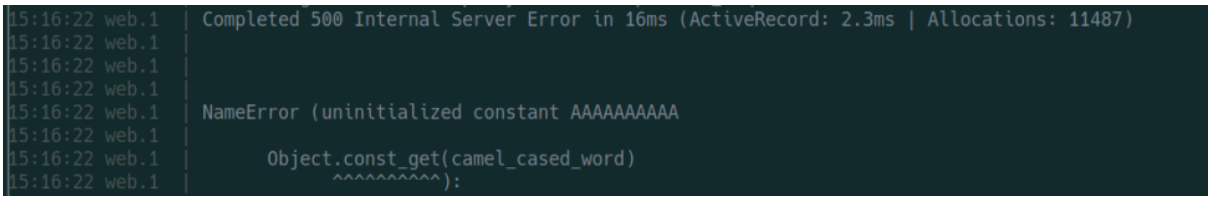
Corrupting the database with unusable classes will cause a crash at the application while viewing the new record or the index view (partial DoS)



Manual delete the corrupted resource in order to recover the applications functionality



Of course it is possible to use other class names or namespaces. The local development environment displays the backend error message when visiting a corrupted record. Avo is trying to apply a scope to this class that does not exist.



Specifying an invalid class name in the parameter will cause the application to crash again while trying constanize the provided string

Rating

The final exploitation of this vulnerability requires more time than is provided in this assessment, but initial testing of the post request shows the potential critical risk. The classes could be instantiated at any point in the code and this could also lead to code execution.

Recommendation

Avo should be configured to never trust user-supplied input, especially when defining classes for records. In this particular case, Avo can evaluate the options list given for the polymorphic field and only allow strings from that list. With this white-list approach, an attacker cannot supply unintended classes.

Code smell

low

Description

While reviewing the source code of the target application, some non-best practices / potential security issues were identified that should be addressed to avoid introducing new vulnerabilities.

app/components/avo/field_wrapper_component.html.erb

```
32    <% if params[:avo_debug].present? %>
33      <!-- Raw value: -->
34      <!-- <%== field.value.inspect %> -->
35    <% end %>
```

app/components/avo/index/field_wrapper_component.html.erb

```
20    <% if params[:avo_debug].present? %>
21      <!-- Raw value: -->
22      <!-- <%== @field.value.inspect %> -->
23    <% end %>
```

Rating & Recommendation

Even if there is no impact, the use of user-controlled input to switch to debugging features can introduce unexpected vulnerabilities.