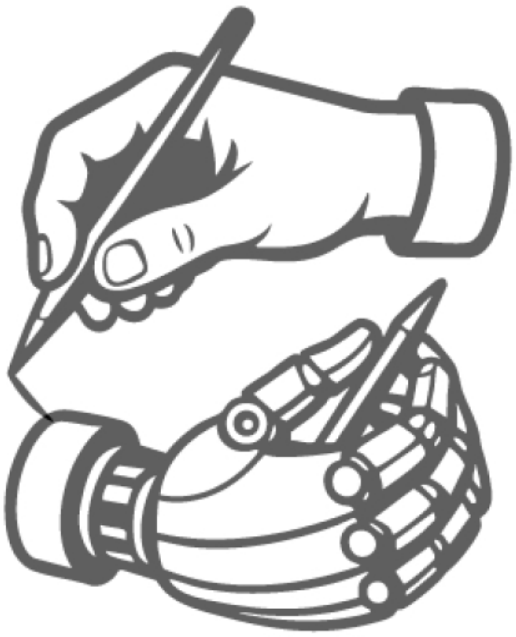

Hacktron x Gumroad

Autonomous Security Assessment- 04.2025

Presented By:
Hacktron AI, M. Pedhapati, H. Jaiswal, Z. Zhang

Contents

Executive Summary

This document provides a comprehensive overview of the security assessment conducted on the Gumroad Application using Hacktron's autonomous AI agents.

As part of our broader research initiative to evaluate AI capabilities in real-world offensive security, the Hacktron AI Lab developed practical benchmarks HackBench and autonomous security agents Hacktron. In April 2025, the security research team at Hacktron conducted a white-box security assessment targeting the Gumroad Application, along with other research projects.

This assessment was specifically designed to identify vulnerabilities in the system, with particular focus on the frontend UI and backend APIs. Spanning over 100 million tokens of usage by the agents, the evaluation aimed for comprehensive coverage.

The assessment comprised a single work package:

- **WP1: White-box auditing of the Gumroad Application**

Hacktron autonomously identified a total 12 issues during the assessment, which were subsequently triaged by the Hacktron research team. The triage process filtered out the false positives. After validation, eight findings were classified as security vulnerabilities, while four were categorized as general weaknesses with low exploitation potential.

The detailed findings are organized chronologically in the following sections, beginning with identified security vulnerabilities, followed by general weaknesses. Each finding includes a technical breakdown, proof-of-concept demonstrations where applicable, and recommended remediation strategies.

The report concludes with reflections from the Hacktron AI Lab team on Hacktron's autonomous capabilities, outcomes observed during the assessment, and the future scope for advancing autonomous offensive security research.

Scope

- **Autonomous code audits against Gumroad Application, UI & APIs**
  - **WP1: White-box auditing against Gumroad Application, UI & APIss**
    - **Sources**
      - https://github.com/antiwork/gumroad

Findings

The findings have been categorized into two distinct groups: Vulnerabilities and Miscellaneous Issues. Vulnerabilities, characterized by their immediate impact, necessitate urgent remediation. Conversely, miscellaneous issues, while lacking immediate consequences, play a crucial role in proactively mitigating potential future vulnerabilities.

Vulnerabilities

In this section, an in-depth technical analysis is presented for vulnerabilities discovered during the penetration test. Given their immediate security implications, we strongly recommend swift remediation. The severity of each issue is encapsulated within square brackets

GUM-01-001 WP1 [High]: DOM XSS via Unsafe innerHTML Assignment in Tiptap Raw Node

Fix Note: This issue was fixed and the fix was verified by Hacktron. The documented problem no longer exists.

During the code review, Hacktron identified a Cross-Site Scripting (XSS) vulnerability within the profile editing functionality. The vulnerability arises due to unsafe handling of user-supplied HTML in the Tiptap editor's *Raw* extension, specifically in the *Raw.renderHTML* function.

User input containing a *<div class="tiptap__raw">[MALICIOUS_HTML]</div>* element is processed in the Profile Editor (*EditSections.tsx*) and parsed by the *Raw.parseHTML* function. The malicious HTML is extracted via *innerHTML* and stored in the *html* attribute of the Raw node in the editor's JSON state. This unsanitized JSON is subsequently saved to the backend through *ProfileSectionsController#create/update*, passing through *SaveContentUpsellsService#from_rich_content* without filtration.

When rendering the profile page, the *Raw.renderHTML* function sets *doc.innerHTML* directly using the previously stored, untrusted HTML. This leads to immediate JavaScript execution in the context of any visitor viewing the crafted profile.

Affected files:
/app/app/javascript/components/Profile/EditSections.tsx
/app/app/javascript/components/RichTextEditor.tsx
/app/app/javascript/components/TiptapExtensions/MediaEmbed.tsx

Affected cpde:
innerHTML assignment in *Raw.renderHTML* (MediaEmbed.tsx:67)

Impact: An authenticated user with profile editing permissions can inject arbitrary JavaScript, leading to session hijacking, phishing attacks, or further client-side compromise affecting all visitors to the profile page.

PoC:

```
<div class="tiptap__raw">
  <img src=x onerror=alert('XSS-ProfileEditor-via-RawNode')>
</div>
```

Full POC created by Hacktron Researcher:

```
<!-- https://siriusly4.gumroad.com/ csp bypass -->
div class="tiptap__raw"><script src="https://www.google.com/
   complete/search?client=chrome&q=123&jsonp=alert(1337)//"><
   /script>
/div>
```

Root Cause: The application directly uses untrusted HTML from user input without sanitization when assigning to *doc.innerHTML* in the *Raw.renderHTML* method.

Remediation:

- **Sanitize HTML:** Modify *Raw.renderHTML* to sanitize the *HTMLAttributes.html* content before assigning it to *doc.innerHTML*, using a robust HTML sanitizer.
- **Restrict Extension:** Remove the *Raw* extension from *baseEditorOptions* in *RichTextEditor.tsx* unless strictly necessary. If needed, ensure all outputs from *Raw* nodes are sanitized.
- **Backend Sanitization (Defense-in-Depth):** Implement sanitization within *SaveContentUpsellsService#from_rich_content* to clean any *html* attributes stored in Tiptap JSON data.

GUM-01-003 WP1 [High]: DOM-based XSS via iframe.ly Embed Handling in MediaEmbed.tsx

Hacktron identified a DOM-based Cross-Site Scripting (XSS) vulnerability in the media embedding workflow of the Tiptap editor. The vulnerability arises when untrusted HTML received from the external *iframe.ly* service is injected into the DOM via *dangerouslySetInnerHTML* or *innerHTML*, without sanitization.

Component:
/app/app/javascript/components/TiptapExtensions/MediaEmbed.tsx

Impact:
If an attacker can craft a URL that causes *iframe.ly* to return malicious HTML (e.g., with a *<script>* tag), that HTML will be injected into the DOM and executed in the victim's browser. This could lead to full session compromise, phishing, or other client-side attacks.

Root Cause:
The application trusts and directly injects HTML from iframe.ly based on user-controlled input, without applying any post-response sanitization. This leads to XSS if iframe.ly returns attacker-controlled content.

Hacktron Researcher PoC:
The retrieved HTML is not sanitized and stored directly in backend, we can update it via following request.

```
POST /links/upihb HTTP/2
Host: gumroad.com
Cookie: _gumroad
Content-Type: application/json

{
  "name": "asd",
  "is_published": true,
  "rich_content": [{
    "id": "DYRSDaOriagyweVovv8Yig==",
    "page_id": "DYRSDaOriagyweVovv8Yig==",
    "description": {
      "type": "doc",
      "content": [{
        "type": "mediaEmbed",
        "attrs": {
          "url": "https://www.youtube.com/watch?v=-qJatoCWZGE
   ",
          "html": "<script src=\"https://www.google.com/
   complete/search?client=chrome&q=123&jsonp=alert(1337)//\"
   ></script>",
          "title": "Visualizing MLP learning dynamics"
        }
      }, {
        "type": "paragraph"
      }]
    }
  }]
}
```

PoC:

`https://gumroad.com/d/f9a7d53246cf5efb0c784f83cf81e97c`
Use: `a@a.com` to trigger the payload.

Remediation:

- **Avoid Direct Rendering:** Do not inject untrusted HTML from external services using `dangerouslySetInnerHTML` or `innerHTML`.
- **Sanitize iframe.ly Output:** If rendering is necessary, apply HTML sanitization using a library such as `DOMPurify`, ensuring all scripts, event handlers, and dangerous tags are removed.
- **Use Sandboxed Iframes:** Instead of rendering third-party embeds inline, use sandboxed `<iframe sandbox>` to contain untrusted content safely.

GUM-01-004 WP1 [High]: Stored XSS via Product Description Rendering

Hacktron identified a Stored Cross-Site Scripting (XSS) vulnerability in the product description handling logic. The vulnerability stems from the lack of sanitization in the backend processing pipeline when saving or rendering product descriptions.

Component:
/app/app/javascript/components/Product/index.tsx

Impact:
An authenticated seller can insert malicious HTML or JavaScript into a product's description field. This content is rendered via *dangerouslySetInnerHTML* on public product pages, resulting in XSS when any visitor loads the page. The attack could be leveraged to steal session tokens, perform phishing attacks, or hijack user accounts.

Root Cause:
The backend does not sanitize arbitrary HTML content in the description field before saving or rendering. Only specific `<public-file-embed>` tags are validated, leaving other embedded JavaScript untouched.

Hacktron Researcher PoC:

```
POST /links/ro HTTP/1.1
Host: gumroad.dev
Content-Type: application/json

{
  "name": "test",
  "description": "<p>hiefindme<img src=x onerror=alert(1) /><
   script src='https://cdn.iframe.ly/api/iframely/?url=https
   ://google.com&api_key=6317bed3ca048a1a75d850&import=0&
   callback=alert&format=xml'></script></p>",
  "is_published": true,
  ...
}
```

PoC by Hacktron researcher:
`https://9912484174829.gumroad.dev/l/ro`

Remediation:
- **Backend Sanitization:** Sanitize the product description field using a robust HTML sanitizer (e.g., `rails-html-sanitizer`) inside *SavePublicFilesService* or *LinksController#update*.
- **Frontend Defense-in-Depth:** Ensure that the `product.description_html` passed to the frontend is generated via a fully sanitized or Markdown-based renderer.
- **Avoid dangerous rendering:** Prefer not using `dangerouslySetInnerHTML`; alternatively, move to safer rendering strategies (e.g., Tiptap's `EditorContent`) whenever possible.

GUM-01-005 WP1 [High]: Stored XSS via Seller Display Name in Receipt Generation

Hacktron identified a Stored Cross-Site Scripting (XSS) vulnerability within the receipt generation flow affecting both email and web-based receipt views. The vulnerability stems from improper handling of seller-provided names, leading to unsanitized HTML rendering.

Components:
/app/app/models/user.rb
/app/app/presenters/receipt_presenter/charge_info.rb
/app/app/views/customer_mailer/receipt/sections/_items.html.erb

Impact:
An attacker controlling a seller account can inject malicious JavaScript into their profile 'Name' field. Upon purchase, the malicious content is embedded into the customer's receipt (both in emails and web views) and executed within the user's browser context. This could result in session hijacking, phishing attacks, or complete account compromise for buyers.

Root Cause:
User-controlled input (seller display name) is interpolated into HTML content without escaping, combined with explicit invocation of `html_safe`, leading to a Stored XSS condition.

PoC:

```
Display Name:
<iframe srcdoc=<script src=//www.google.com/complete/search?
   client=chrome&callback=alert#"></scrip
```

PoC URL:
`https://gumroad.com/purchases/7AhL7umVp-mIUs8gsBBmLw==/receipt?email=a%40a.com`

Additional Note by Researcher: The payload had to be crafted carefully to bypass CSP restrictions and remain under 100 characters, confirming the exploitability under real-world constraints.

Remediation:

- **Escape User Input:** Properly HTML-escape the seller's display name before interpolation into any HTML string inside `charge_info.rb`.
- **Remove html_safe Usage:** Avoid using `html_safe` unless absolutely necessary and safe.
- **Escape in Views:** Alternatively, ensure that `charge_info.product_questions_note` is escaped using `h()` when rendered in views.

GUM-01-007 WP1 [Critical]: SQL Injection in ORDER BY Clause via Unvalidated sort_direction

Fix Note: This issue was fixed and the fix was verified by Hacktron. The documented problem no longer exists

Hacktron identified a critical SQL Injection vulnerability affecting the ORDER BY clause construction in the UTM Links API. The sort_direction parameter, controlled via the request query, was directly interpolated into an SQL fragment without proper allow-list validation.

Affected file:
/app/app/presenters/paginated_utm_links_presenter.rb (Line 63)

Vulnerable Code Snippet (Line 63):

```
order(Arel.sql("\#{sort\_key} \#{sort\_direction}"))
```

Impact:
An attacker can exploit this vulnerability to inject arbitrary SQL into the ORDER BY clause. Depending on the database in use and permissions granted, this could enable time-based blind SQL injection, data exfiltration, service denial, or even lateral database attacks.

Proof of Concept:

- Request:

```
GET /api/internal/utm_links?sort[key]=created_at&sort[
   direction]=desc,(SELECT CASE WHEN (1=1) THEN SLEEP
   (5) ELSE SLEEP(0) END)
```

- Expected Outcome: A 5-second delay in the server response time if the injection succeeds (assuming MySQL/MariaDB backend).

Remediation:
- Validate the `sort_direction` parameter explicitly against an allow-list of `['asc', 'desc']`.
- Default to a safe value (e.g., 'asc') if the provided `sort_direction` is invalid or missing.
- Consider raising an exception or rejecting the request entirely if an unexpected value is encountered.
- Avoid interpolating user-controlled input directly into `Arel.sql` fragments wherever possible.

GUM-01-008 WP1 [Low]: IDOR in Email Unsubscribe Functionality

Hacktron identified a potential Insecure Direct Object Reference (IDOR) vulnerability within the email unsubscribe feature implemented in the `UsersController`.

Affected file:
/app/app/controllers/users_controller.rb

Vulnerable Code Snippet:

```
def email_unsubscribe
  @action = params[:action]

  if params[:email_type] == "notify"
    @user.enable_payment_email = false
  elsif params[:email_type] == "seller_update"
    @user.weekly_notification = false
  elsif params[:email_type] == "product_update"
    @user.announcement_notification_enabled = false
  end

  @user.save!
  flash[:notice_style] = "success"
  redirect_to root_path
end

private

def set_user_for_action
  @user = User.find_by_external_id(params[:id])
  e404 if @user.nil?
end
```

Impact:
An attacker able to obtain or guess another user's `external_id` could craft requests to unsubscribe that user from email notifications without authorization. This undermines user autonomy over notification preferences and may lead to account disruption.

Proof of Concept:

- An attacker authenticates as their own account.
- Sends a request like:

  ```
  GET /users/email_unsubscribe/4051620356512?email_type=
     product_update
  ```

- Result: The victim's email preferences are changed without consent.

Root Cause:
The system relies solely on `external_id` to locate users without performing an ownership or authorization check.

Remediation:
- Ensure that after locating the user via `external_id`, the application verifies that the located user matches `current_user`.
- If the users do not match, reject the request and return an authorization error.
- Alternatively, avoid accepting `external_id` as a parameter at all for sensitive operations tied to user identity. Always derive user context from the session.

GUM-01-009 WP1 [Low]: IDOR/BOLA in Affiliate Request Approval

Note: Due to the random nature of the `external_id` format, the probability of a successful attack is low in practice without information leakage. However, fixing the authorization gap remains important to prevent future abuse.

Hacktron identified an Insecure Direct Object Reference (IDOR), also known as Broken Object Level Authorization (BOLA), in the affiliate request approval flow.

Affected file: */app/app/controllers/affiliate_requests_controller.rb*

Vulnerable Code Snippet:

```
before_action :set_affiliate_request, only: %i[approve ignore
   ]

def set_affiliate_request
```

```
  @affiliate_request = AffiliateRequest.find_by_external_id!(
   params[:id])
end

def approve
  perform_action_if_permitted do
    AffiliateRequests::ApproveService.call(@affiliate_request
   )
    respond_successfully
  end
end
```

Impact:
An unauthenticated attacker who knows or can guess a valid `external_id` can approve any pending affiliate request by sending a crafted GET request. Although external IDs are designed to be random and hard to guess, the absence of authorization checks represents a security weakness.

Proof of Concept:

- Access the following URL without authentication:

  ```
  https://gumroad.com/affiliate_requests/
     mLKaMl99F5IN3Uv9EFvS9g==/approve
  ```

- If the external ID corresponds to a pending affiliate request, it will be approved immediately.

Root Cause:
The application trusts user-supplied `external_id` input without ensuring that the acting user owns or is authorized to modify the referenced resource.

Remediation:

- Require authentication for the `approve` action by removing it from `PUBLIC_ACTIONS`.
- Scope the affiliate request lookup to the authenticated seller, e.g., `current_user.affiliate_requests.find_by_external_id!(params[:id])`.
- Alternatively, use a dedicated authorization mechanism such as `Pundit` or `Cancancan` to enforce access control.

GUM-01-010 WP1 [Low]: IDOR in Mobile Preorder Attributes API with Hardcoded Mobile Token

Fix Note: Hacktron discovered a potential IDOR in the mobile preorder attributes API. This turned out to be a false positive, while triaging this security researcher noticed that the mobile token is hardcoded.

Hacktron identified an Insecure Direct Object Reference (IDOR) vulnerability in the `api/mobile/preorders#preorder_attributes` endpoint.

Affected files:
/app/app/controllers/api/mobile/preorders_controller.rb
/app/app/controllers/api/mobile/base_controller.rb

Note:
This was a false positive by Hacktron, While triaging our researcher found that the API relies on a hardcoded static token (`MOBILE_TOKEN`) embedded in the source code:

```
MOBILE_TOKEN = "ps407sr3rn[..snip..]9r5469ososoo"
```

Possession of this mobile token allows public access to any mobile API endpoint protected only by this token, amplifying the risk of unauthorized access.

Impact:
An attacker possessing a valid `external_id` for a preorder in `authorization_successful` or `charge_successful` state can access preorder attributes without proper ownership verification. Sensitive information, such as the `user_id` and `purchase_id`, could be leaked to unauthorized users.

Proof of Concept:

- Assume attacker knows a valid external ID `abc123xyz`.
- Perform the following request:

```
curl -X GET "https://gumroad.com/mobile/preorders/
    preorder_attributes/abc123xyz" \
-H "Authorization: Bearer [attacker's_auth_token]"
```

- The server responds with preorder details, including `user_id` and `purchase_id`.

Remediation:
- Implement ownership verification: Ensure that the fetched preorder belongs to the currently authenticated user before exposing any attributes.
- Deprecate the use of hardcoded mobile tokens. Implement a proper authentication flow (e.g., OAuth2, JWT) for mobile APIs.
- Rotate the leaked mobile token immediately and treat all endpoints relying solely on this token as compromised.
- Review all mobile API endpoints for similar authorization gaps.

Miscellaneous Issues

In this section, we discuss findings that, although they did not lead to immediate exploitation, have the potential to assist an attacker in achieving their malicious goals in the future

GUM-01-002 WP1 [Info]: Weak Host Validation in isValidHost Function

During the code review, Hacktron identified weak validation logic within the *isValidHost* function located in */app/app/javascript/widget/utils.ts*.

The function attempts to validate the origin of incoming *postMessage* events by checking whether the *url.host* string *endsWith* the *ROOT_DOMAIN* or a *customDomain*. However, this approach is inherently insecure because it fails to account for boundary conditions. An attacker could register a malicious domain such as *attacker.com.trusted.com*, which would incorrectly pass the validation intended for *trusted.com*.

Affected file:
/app/app/javascript/widget/utils.ts

Affected function:
isValidHost

Impact: Currently, the message handlers that rely on *isValidHost* (located in *embed.ts* and *overlay.ts*) only interact with relatively safe sinks such as *style.height* and *ariaLabel*. Therefore, no immediate security risk such as XSS was observed.

However, the weak validation approach introduces a latent risk. If future code changes introduce message handlers that forward message data into more dangerous sinks (e.g., *innerHTML*, *eval*, *location.href*), this validation flaw could facilitate serious vulnerabilities, including Cross-Site Scripting (XSS) or Open Redirects.

Root Cause:
The validation logic only checks if the domain *endsWith* a trusted domain string, without verifying domain boundaries (e.g., ensuring a separating dot character or performing exact matches where necessary).

Remediation:
- Refactor *isValidHost* to perform stricter validation, ensuring that only legitimate domains are accepted.
- Use exact string matching against a list of trusted origins where possible.
- Ensure domain boundary checks, verifying that the character preceding the domain suffix is a dot (e.g., "`.trusted.com`").
- If subdomains are unnecessary, compare origins using strict equality (===) against *process.env.ROOT_DOMAIN*, *process.env.SHORT_DOMAIN*, and *customDomain*.

GUM-01-006 WP1 [Info]: Stored XSS via Unsanitized Third-Party Analytics Snippets

Note: This is considered miscellaneousissue as code executes inside a sandboxed domain (a separate origin from gumroad.com)

Hacktron identified a stored XSS vulnerability in the third-party analytics settings workflow. Sellers were permitted to input arbitrary HTML and JavaScript into the analytics configuration form. This untrusted code was stored and later rendered using the `raw` helper, without sanitization.

Steps to reproduce:
- Sellers visit `/settings/analytics` and input analytics code in the form.
- The `Settings::ThirdPartyAnalyticsController#update` action permits the `code` parameter.
- The `ThirdPartyAnalytic.save_third_party_analytics` method stores the provided code directly to the `analytics_code` field.
- On page views that load analytics (e.g., product pages), the code is fetched by `ThirdPartyAnalyticsController#index`.
- The code is injected using the `raw` helper inside `third_party_analytics/index.html.erb`, leading to execution.

Impact:
This allows a seller to persistently inject JavaScript that will run whenever pages tied to that analytics code are loaded. The implications include session hijacking, cookie theft, phishing, or other attacks against buyers and administrators.

Affected files:
- `/app/app/controllers/settings/third_party_analytics_controller.rb`
- `/app/app/models/third_party_analytic.rb`
- `/app/app/controllers/third_party_analytics_controller.rb`
- `/app/app/views/third_party_analytics/index.html.erb`

Remediation:
- Sanitize the `analytics_code` field before saving, using a strict allowlist approach to prevent execution of scripts or event-based attributes.
- Avoid using the `raw` helper directly with user-supplied content.
- Consider rendering third-party code inside a sandboxed `<iframe>` or injecting it client-side with CSP isolation if required for compatibility.

GUM-01-011 WP1 [Low]: Unauthenticated Purchase Unsubscribe via IDOR in PurchasesController

Note: Given that exploitation depends on acquiring valid `external_id` values, the practical risk is currently considered low. However, reliance on security-by-obscurity (randomness of IDs) can become fragile if other parts of the system accidentally expose these identifiers.

Hacktron identified an Insecure Direct Object Reference (IDOR) vulnerability in the `PurchasesController#unsubscribe` action.

Affected files:
/app/app/controllers/purchases_controller.rb

Impact:
Any user who knows the external ID of a purchase record can trigger the unsubscribe flow for that purchase, causing the legitimate buyer to be unsubscribed from seller communications. This operation does not require authentication or authorization verification, relying solely on possession of the external ID.

Proof of Concept:
An attacker obtains a valid `external_id` (e.g., through URL leakage) and sends a request:

```
curl -X DELETE "https://gumroad.dev/purchases/4Dk[..snip..]5
   T1xA==/unsubscribe"
```

The associated purchase record will be unsubscribed without verifying the requester's identity.

Additional Observations:
Multiple actions in `PurchasesController` (such as `receipt` and `generate_invoice`) are also public and rely on knowledge of the `purchase_id` and associated `email` for access. While currently not critical, any leakage of purchase IDs externally could expose sensitive metadata. If intended, this model should be documented and revisited periodically for risk assessment.

Remediation:
- Remove `unsubscribe` from `PUBLIC_ACTIONS` and enforce authentication, ensuring `current_user` matches the purchaser.
- Alternatively, if unauthenticated unsubscribe is required (e.g., for email links), implement signed tokens (e.g., expiring HMAC-based URLs) to validate legitimacy instead of relying only on the external ID.
- Review all `PurchasesController` endpoints that are public and reassess whether email + purchase ID combinations are sufficient protection against enumeration or leakage.

GUM-01-012 WP1 [Low]: Potential XSS via Arbitrary HTML Upload to files.gumroad.com

Fix Note: This was manually identified by Hacktron researchers during triage. It was not autonomously detected by Hacktron agents. Although no direct exploit path was confirmed, proactive hardening is advisable.

Hacktron researchers observed that the `generate_multipart_signature` functionality in the S3 utility controller allows sellers to upload arbitrary files, including files with a `text/html` MIME type.

Affected file:
/app/app/controllers/s3_utility_controller.rb

Proof of Concept:
An example request to sign and upload a malicious HTML file:

```
GET /s3_utility/generate_multipart_signature?to_sign=POST%0a
   %0atext%2fhtml%0a%0ax-amz-acl%3aprivate%0ax-amz-date%3aMon
   %2c%2028%20Apr%202025%2013%3a55%3a27%20GMT%0a%2fgumroad%2
   fattachments%2f4523355617373%2fs%2foriginal%2fvideo-review
   .html%3fuploads HTTP/2
```

If later this file is embedded or linked within the Gumroad ecosystem without proper sandboxing or content-disposition headers (e.g., via signed URL listing, previews, etc.), malicious JavaScript could execute in the context of the victim's browser.

Impact:
While no active exploitation vector was identified during this assessment, the ability to upload arbitrary `text/html` files into a trusted Gumroad domain represents a latent risk. Future features (e.g., file previews, direct file serving) could inadvertently expose users to Stored XSS.

Root Cause:
- Insufficient restriction on allowable MIME types during multipart signature generation for seller uploads.
- Lack of enforcement for safe Content-Types (e.g., `application/octet-stream` for unknown uploads).

Remediation:
- Enforce stricter content-type allowlists at the time of signature generation (e.g., only permit known-safe types like images, PDFs, videos).
- Ensure that all served uploaded files enforce secure `Content-Disposition: attachment` headers unless explicitly intended to be displayed inline.
- Consider serving uploaded attachments from an isolated, sandboxed domain separate from Gumroad's main web application domain to prevent privilege escalation in the event of XSS.

Conclusions

At Hacktron, our vision extends far beyond traditional security testing. We are pioneering research into autonomous offensive security — building agents capable of learning, reasoning, and finding vulnerabilities across the critical software ecosystems that power the internet.

Our ambition is bold: to create systems that can independently discover vulnerabilities in major open-source projects like Apache HTTP Server, Chromium, Linux, and other foundational infrastructure at internet scale.

In pursuit of this vision, we developed Hacktron: a coordinated system of AI agents designed to autonomously perform source code analysis, dynamic application security testing, and vulnerability triage. These agents are trained not merely to scan, but to understand complex codebases, reason about security flaws, and prioritize findings with real-world impact.

As part of our ongoing research, Hacktron was deployed against the Gumroad codebase, where it autonomously uncovered 11 vulnerabilities, including four rated as high or critical severity.

The Gumroad team responded swiftly, addressing the critical findings and validating the effectiveness of proof-driven autonomous security discovery.

Our work is only beginning. We are actively advancing toward agents capable not only of identifying vulnerabilities but also autonomously suggesting — and eventually implementing — sophisticated patches. In parallel, we are establishing open, transparent benchmarks(Hackbench) to objectively measure AI's true capabilities in real-world offensive security contexts.

If you are interested in receiving a free security audit, collaborating on our research initiatives, or supporting the development of autonomous security agents for the public good, you can request an audit through this form. For further queries, please reach out to us directly at hello@hacktron.ai