Hacking Gemini: A Multi-Layered Approach

Valentino Massaro - valentino@buganizer.cc - @valent1nee

Table of contents

bugSWAT Tokyo

In April 2025, I had the opportunity to attend bugSWAT (live hacking event) organized by Google VRP in Tokyo, where we focused on hacking Google AI products. You can read more about it here: AI bugSWAT in Tokyo & 2025 Hacker Roadshow.

Before the in-person event, I had 2 weeks to find security issues in the presented scope, which was a real challenge. Fortunately, I already had some experience looking for bugs in one product, and already understood various things that could result in high-impact security issues. During those long nights, I spent my time re-reading awesome research written by Orange Tsai and James Kettle, looking for inspiration and sparking new ideas

related to multi-layered architecture and discrepancies. And when I had to focus on hunting, I listened to the Critical Thinking Podcast (hosted by Rhynorater, with co-hosts rez0 and gr3pme) which helped me stay motivated.

In this exciting event, I met with 14 top researchers from Google VRP at the Google office located in Shibuya Stream. I had plenty of fun in the 2-day in-person hacking event. After the event, we rented an entire office room in Starbucks (didn't know it was possible) where we had really exciting conversations with Alesandro, Liv, Sivanesh and Sreeman -extremely talented and friendly hunters in Google VRP-, and we also visited Akihabara to play arcade games =)

This event was a dream come true. Being invited to Tokyo, talking with peers about what we are passionate about, and finally finding some decent bugs in a Google live hacking event after starting to understand how to approach these complex multi-layered products. In this case, I was able to accomplish 2nd place for various findings in Gemini, where I decided to delve into deeply...

I want to thank Gábor Ács-Kurucz for being a mentor and a great engineer to work with while improving the security of Gemini. Thanks to Martin Straka, Sam Erb, Jan Keller, and all the VRP members involved, for organizing bugSWAT events and ensuring the VRP provides a great researcher experience –always listening to feedback and improving these events year after year.

Anyway, I hope to keep finding nice bugs in Google :)

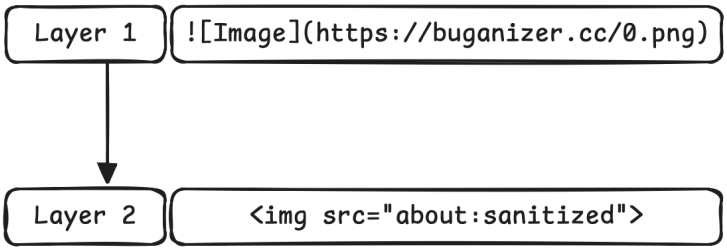
Gemini markdown sanitizer bypass

Introduction

Between 2023-2024, after reading the latest research on vulnerabilities in LLM chat applications, I noticed most of them used Markdown images to perform data exfiltration. However, Markdown image injections were fixed in Gemini due to multiple reports from different researchers leaking Workspace data:

- https://embracethered.com/blog/posts/2023/google-bard-data-exfiltration/
- https://www.landh.tech/blog/20240304-google-hack-50000

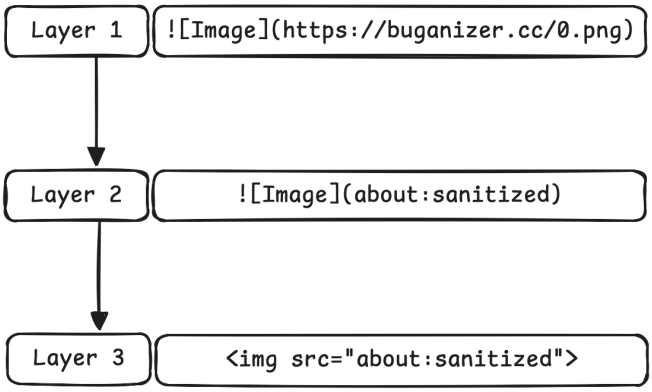
After the implemented fixes, if you were to ask Gemini to return a Markdown image, the following would occur.



At first, it looked safe enough, but after experimenting a lot with the application, I discovered some interesting patterns.

Image rendering process

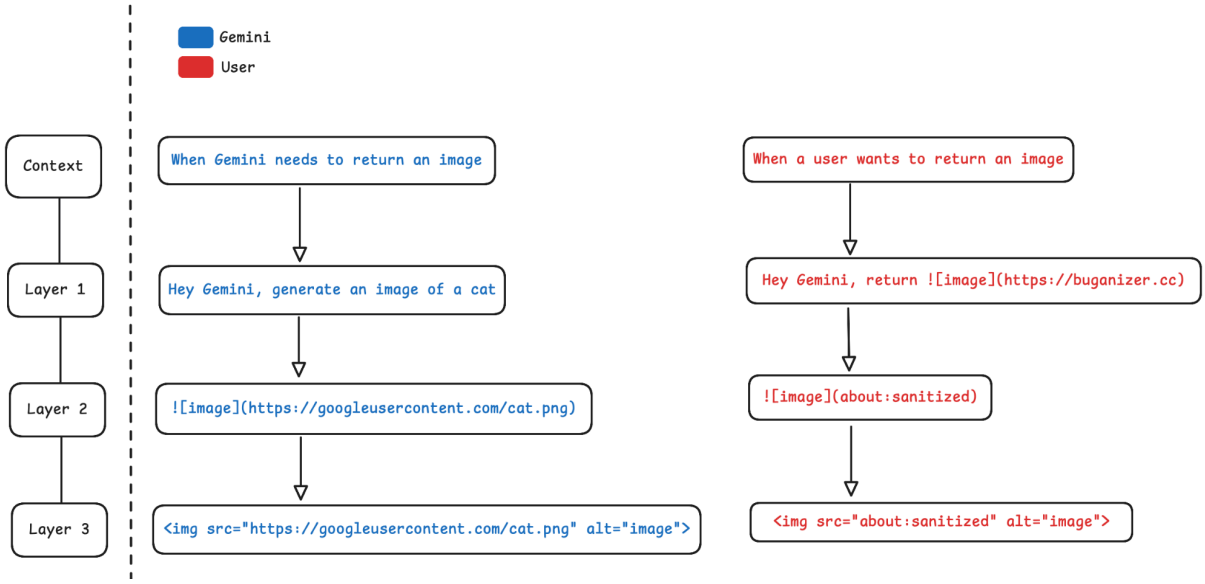
Although the image rendering process I described seems simple (just converting a Markdown image to HTML), in that phase of the research, I was missing an intermediary layer. Instead, the process looked more like this:



Something strange was happening, and it had to have a reasonable explanation, but at that moment, it was just an inexplicable feeling. In my mind, I wondered why images were still allowed at all, but a more subtle detail was that they were sanitized before being converted to HTML. So, I thought about potential reasons for this behavior, and had an idea which relates to something like "when they need to use something (e.g., images) that would otherwise be disallowed or inaccessible... they implement things in a particular way".

In the following diagram, you can notice two different processes related to images.

- The first process occurs when Gemini needs to generate and return an image whenever a user requests it. In this case, Gemini doesn't trust user-supplied input from Layer 1, but it concatenates a trusted Markdown image in Layer 2, that is later rendered in Layer 3 as HTML.
- The second process occurs when a user wants to return a Markdown image by specifying it in Layer 1, which is treated as untrusted input. Therefore, that content is sanitized before rendering occurs in Layer 3.



With those processes in mind, let's hold that thought for a moment (it might be useful later), and try to continue analyzing the application.

Linkification process

When you send instructions to Gemini, those are sent in plaintext, therefore, there is a need to identify and linkify segments of text that represent hyperlinks. For instance, google.com/ would be linkified using google.com, and then rendered as google.com/. What piqued my curiosity here is the 3-layer approach (instead of 2-layer), which seemed to increase complexity in linkifying URIs across different contexts (HTML and Markdown).

Interestingly, // was treated as text, while google.com/ was inserted into the Markdown hyperlink using https:// as a prefix.

It seemed like certain characters weren't considered part of the hostname when linkifying.

Bypassing the sanitizer

Do you remember the thought that was presented earlier? Well, it might be useful in this case, because if the server wants to return an image for specific purposes, it can be rendered in the 3rd-layer. Therefore, we can inject ! as a prefix in the 2nd-layer and bypass the sanitization. This works because ! is not treated as part of the hostname when linkifying, so it isn't included in the resulting URI. Additionally, since images and hyperlinks in Markdown are syntactically identical except for the leading !, controlling the Markdown hyperlink prefix is enough to switch a link into an image.

Bypassing CSP

There are many open redirects in *.google.com that can bypass CSP :)

Timeline

- Mar 30, 2025 09:19AM: Issue reported
- Mar 31, 2025 10:15AM: Triaged by Trust & Safety team (Abuse VRP)
- Mar 31, 2025 06:43PM: 🎉 Nice catch!
- Apr 29, 2025 02:55PM: Duplicate =(

Conclusion

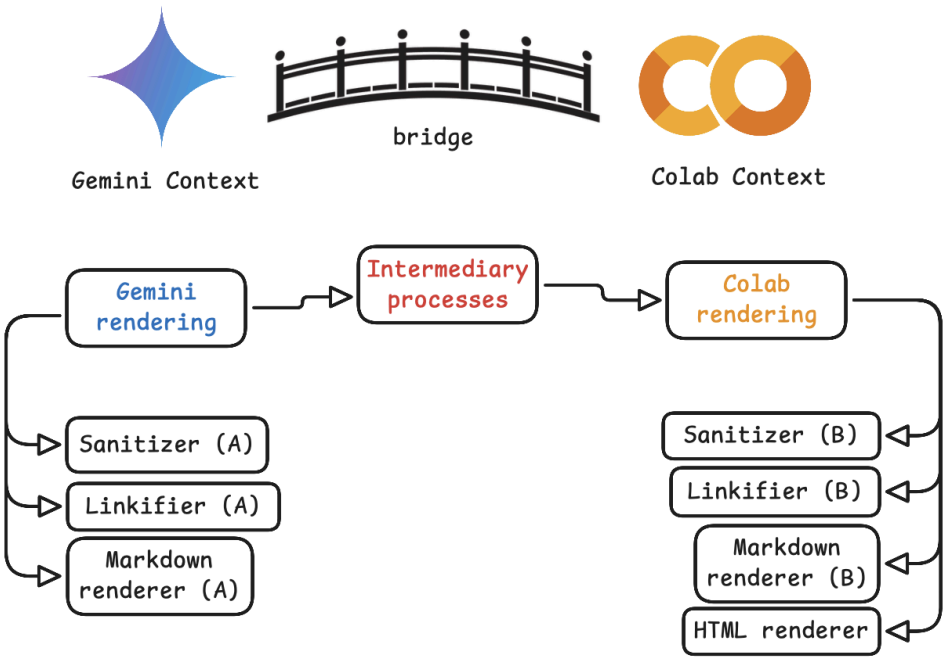
Although it was an already-known issue, finding this bug was a really cool process and gave me many ideas to find other issues later =)

Colaboratory markdown sanitizer bypass

Introduction

Have you looked at my previous issue? (described in Gemini Markdown sanitizer bypass). Well... I was actually looking for bugs in the "Export to Colab" feature before realizing I had found a Gemini sanitizer bypass. It seems that Gemini was performing sanitization before exporting to Google Colab, therefore, the bug would work in both Gemini (0-click) and "Export to Colab" (1-click).

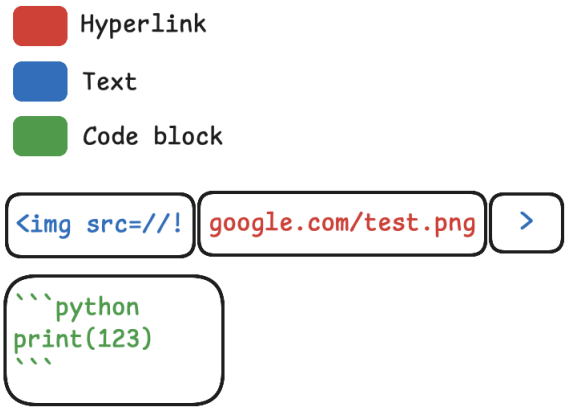
In this context, since Gemini and Colab are applications which parse and render content in a different way, I was looking for discrepancies between them. Generally, features that connect Gemini to other products can be seen as a bridge to other contexts, which expand the attack surface.



The idea of using a bridge might be similar to bridges between deserialization formats; however, in this case it was used to exploit discrepancies between different contexts. A while back, I was reading *Friday the 13th: JSON Attacks (Alvaro Muñoz)*, and the pattern of moving from a context with fewer available exploitation gadgets to one with more

capabilities, such as exploiting a deserialization by creating a bridge between JSON.Net → BinaryFormatter, really inspired me to think about new attack scenarios across different bug classes.

First, the following content was exported to Colab:



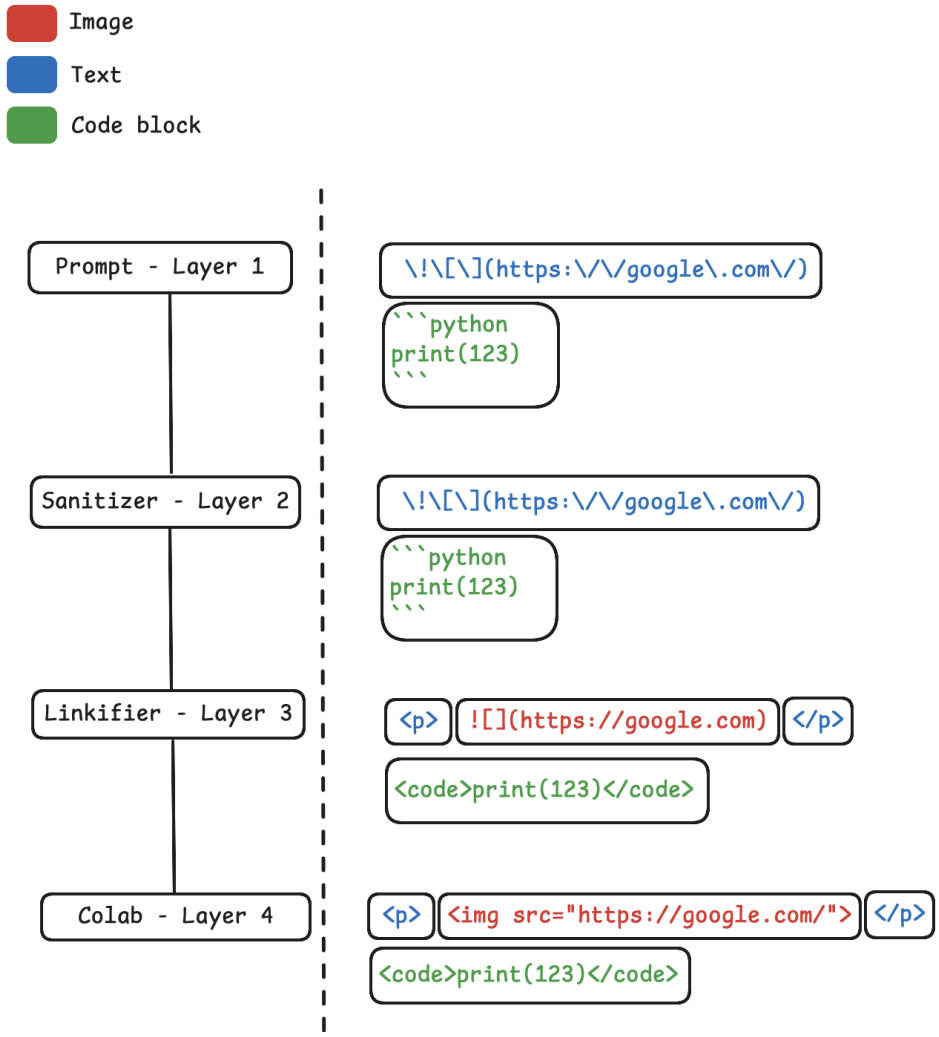
In Colab, it was exported like this:

Escaping the Gemini sanitizer

After the first issue was fixed, the Gemini sanitizer bypass wasn't working anymore, and consequently, it stopped working for Export to Colab as well. After a while, I thought: If I don't want to deal with the Gemini sanitizer, what if I just avoid it?

It turns out that when Gemini doesn't want to interpret something as Markdown, it escapes the content using backslashes.

In this case, we need to include the 4th-layer, which is Google Colab, therefore, the process would look like this:

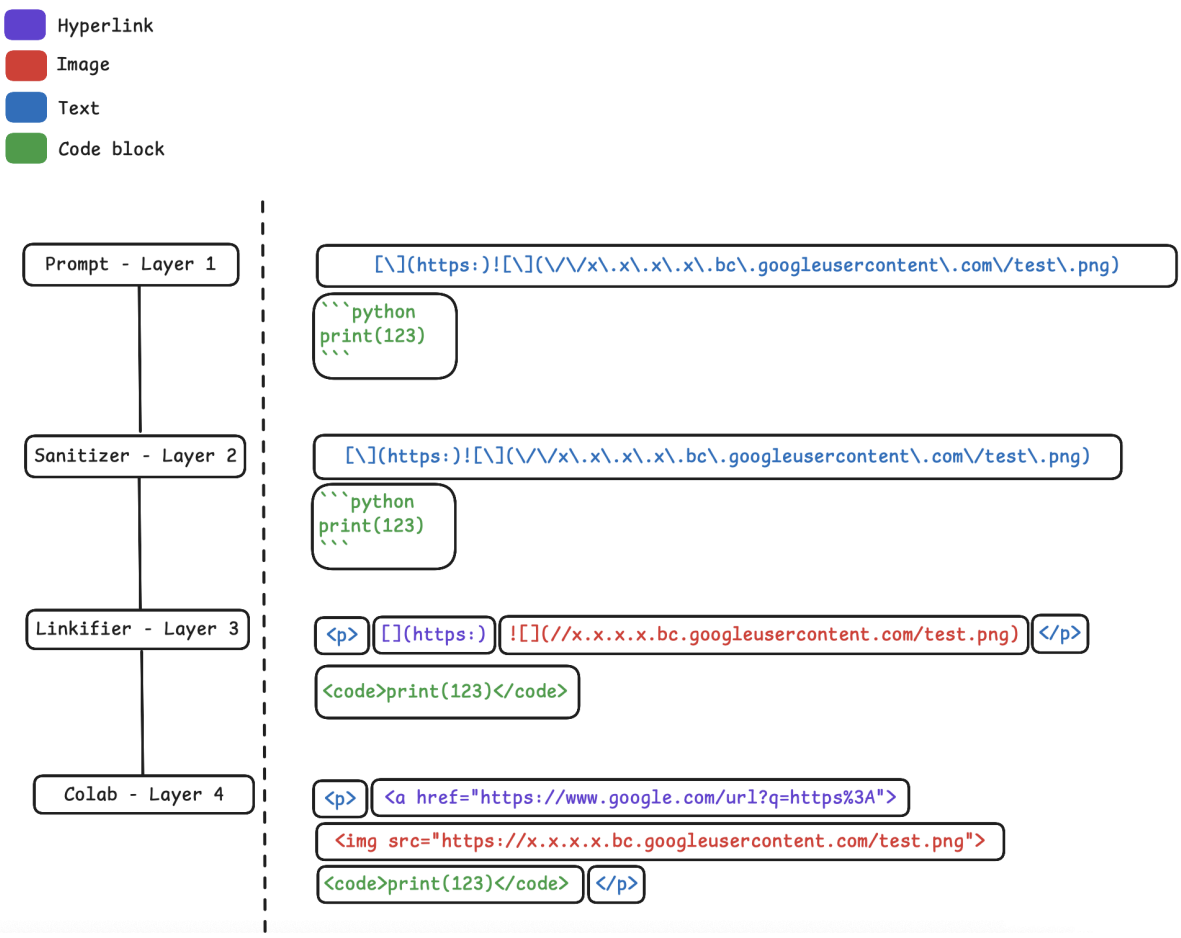


It might be noticed that the URI was also escaped. It was useful to prevent the linkifying process from generating an invalid Markdown image URI such as: <img src='<a href="https://www.google.com/">https://google.com/)'>">.

Confusing Colab

A few days after the issue was reported, before it was reproduced by Google, it suddenly stopped working because something related to escapes seems to have changed in Colab which accidentally broke the chain (probably because there were many layers involved). At that moment, I was finishing my vacation in Puerto Iguazú and was preparing to take the bus to the airport at 6am. So I was really annoyed with Gemini... and I had to find a bypass, it was my mission. Since the bus ride was almost 1hs, I decided to start testing a bunch of ideas in Gemini with my phone, specifically focused on bypasses for that Colab behavior to make the issue fully reproducible again.

After some testing, before arriving at the airport (at 6:30 am approx), I noticed that using an invalid Markdown hyperlink and a Markdown image, confused the process applied in Colab which was preventing the Markdown image injection. When I confirmed the finding, without making too much noise because it was really early in the morning, I said something like: "YESS, THERE YOU GO GEMINI! Now, don't annoy me anymore during my holiday". At that moment, I had to contain myself a bit because it was really early in the morning, but internally I was celebrating like scoring a goal. After that, I recorded a video PoC using a mobile app, and submitted it to the Issue Tracker.



Specifically, it seems [](https:) confused the intermediate process applied when exporting to Colab, preventing the escaping of the Markdown image. When testing, I thought about checking if hyperlink URIs were escaped (they weren't), and what I could achieve with that.

What seemed to happen is that [](https:<unescaped>) wasn't escaped, but was escaped. So, I thought about injecting Markdown into <unescaped> and closing the hyperlink, hoping that with enough luck the intermediary process would be confused and would treat the Markdown image as something that shouldn't be escaped, which would then be rendered correctly in Colab, resulting in the image being rendered.

After arriving home, I re-checked it was possible to leak Workspace data, and recorded another video PoC with a minimal reliable prompt to show the exfiltration vector (what can be fixed consistently in these kinds of issues) on my PC.

Bypassing URI prefix

After trying the prompt a few times to ensure it was reliable, I noticed that sometimes Gemini would linkify and prefix the specified subdomain (e.g., x.x.x.x.bc.googleusercontent.com) with https://www.google.com/search?q=. Therefore, to improve the exploit reliability, I included the userinfo part to bypass the URI parser that was performing that process. Therefore, the exploit involved even more layers!

In the end, the exploit worked successfully bypassing the existing layered protections, and via indirect prompt injection, it was possible to leak arbitrary Workspace data such as Gmail emails, Calendar events, Drive files, and other data retrieved via Gemini extensions.

Timeline

- Apr 30, 2025 07:31PM: Issue reported
- May 5, 2025 01:28PM: Google can't reproduce due to behavior in Colab
- May 6, 2025 11:43AM: Sent variant PoC which bypasses the behavior in Colab
- May 20, 2025 10:57AM: Google Vulnerability Reward Program panel has decided to issue a reward of $20000.00 for your report. Congratulations!

Conclusion

Most delivery methods could work (imported PDF files, emails, etc), and even poisoned training data or embeddings could enable indirect prompt injection. Because IDP (indirect prompt injection) is an unsolved, non-deterministic problem, reports treated the exfil vector as the real issue: it can affect any user who lets Gemini interact with untrusted or poisoned data. Fixing the exfil vector is more tractable (it can have a consistent, deterministic technical fix) while mitigating IDP remains an open challenge.

As AI architectures become more complex and integrated, new attack surfaces emerge. It's reminiscent of the early XSS era -but now you don't necessarily need JS-, because even something that might seem simple, like Markdown, can be an exfil vector.