

APART AI CONTROL SPRINT

Track 3: Building Better Red Teams

When Safety Becomes the Vulnerability:

Base64 Rejection as an Exploitable Attack Vector in Claude

March 2026

Caleb Rudnick & Augusta Lina

Abstract

Any base64-encoded string included in a message to the Claude API causes that request to fail. This behaviour is reproducible across Sonnet and Opus 4.6, and across all platforms including the API, web and mobile applications, and Claude Code. While the blanket rejection of base64 content was originally a reasonable defence against prompt injection—attackers could encode malicious instructions to bypass keyword-based safety filters—the measure has become a liability as large language models have moved from conversational assistants to critical infrastructure components.

This paper presents four controlled experiments demonstrating that the base64 rejection behaviour is reliably exploitable as a denial-of-service and evasion vector. In three of four scenarios, an attacker can crash a Claude-powered service or render specifically targeted content invisible to safety classifiers by simply embedding a base64 string. We further demonstrate a compound attack against a SKILL.md skills database, showing that base64 injection prevents the screening system from evaluating submissions—leaving potentially malicious skills in an unclassified state that downstream agents may treat as safe. These findings illustrate a broader principle in AI control research: that overly rigid safety measures can themselves become attack surfaces, and that robustness requires defences that degrade gracefully rather than fail catastrophically.

1. Introduction

One established method for circumventing LLM safety filters is to base64-encode a malicious instruction before submitting it to the model. Modern large language models possess the ability to decode base64 without external tools, which means that a harmful prompt encoded in base64 can bypass both keyword-based and vector-similarity-based input classifiers. Once decoded by the model, the instruction no longer appears to originate from the user—it appears, in effect, to come from the model itself—which may increase the probability of compliance. For these reasons, treating any message containing base64 as potentially abusive was a rational countermeasure when models were primarily conversational interfaces.

In 2026, however, LLMs are called programmatically in production infrastructure and are applied to millions of inputs at a time. When a model refuses to respond for safety reasons, the downstream consequences can no longer be dismissed as minor inconveniences. A refusal can crash a processing pipeline, leave critical content unclassified, or render information effectively invisible to an organisation's AI-powered systems. The cost of a false positive—rejecting benign input that happens to contain base64—has grown to the point where it can be more damaging than the threat it was designed to prevent.

This paper examines the security implications of Claude's blanket base64 rejection behaviour. We show that the behaviour is not merely an inconvenience but a reliably exploitable vulnerability that can be weaponised for denial of service, safety-filter evasion, and lateral disruption of multi-agent systems. We further demonstrate a compound attack in which base64

injection defeats a purpose-built content screening system, illustrating how excessive caution in one layer can undermine the security of the entire stack.

1.1 Research Scope

This research does not focus on a particular harm category (e.g. generating violent content or enabling self-harm). Instead, it focuses on context manipulation: understanding whether a system prompt, integration architecture, or encoding scheme can suppress Claude’s safety signals or cause catastrophic failure in systems that depend on them. The work falls under Track 3 of the Apart AI Control Sprint—Building Better Red Teams—and aims to surface a class of vulnerability that arises not from model capability but from model over-restriction.

1.2 Threat Model

The attacker is assumed to have no privileged access to the model, its weights, or its system prompt. The only capability required is the ability to inject a short base64 string into content that will eventually be processed by a Claude-powered system. This is a low-skill, low-cost attack: the payload is a single encoded string (e.g. aGVsbG8gd29ybGQ=, decoding to “hello world”) embedded in otherwise ordinary natural-language input. The attacker’s goal varies by scenario but includes crashing services, evading content moderation, and poisoning shared platforms.

2. Methodology

Our experimental design proceeds in two phases. In the first phase, we define and describe the base64 rejection behaviour through simple demonstrations, then construct four controlled experiments that instantiate distinct attack scenarios against realistic integration architectures. In the second phase, we build a SKILL.md screening system and test whether base64 injection can defeat it. All experiments were conducted against claude-sonnet-4-6 via the Anthropic Messages API.

2.1 Define and Describe

We first established two baseline observations. In the first test, a simple statement was base64-encoded and submitted to Claude as a user message. The model returned no response; the safety filters prevented it from processing the input at all. This is distinct from a conversational refusal in which the model explains why it cannot comply—here, the response body is effectively empty, yielding an `IndexError` when the consuming code attempts to access the first element of the content array.

In the second test, a base64 string was embedded within an uploaded document. Claude was again unable to process the document, confirming that the rejection applies not only to direct user messages but to any content that passes through the model’s input pipeline.

2.2 Controlled Experiments

We designed four experiments to map the vulnerability’s attack surface. Each experiment pairs a different attacker goal with a different integration architecture so that the results collectively characterise the range of exploitable scenarios rather than stress-testing a single configuration. The payload used across all experiments was the base64 string `aGVsbG8gd29ybGQ=` (decoding to “hello world”) embedded in otherwise ordinary input.

Experiment 1: Queue Worker

This experiment simulates an AI-powered customer support system in which a Python queue worker iterates over incoming tickets, sends each to the model for categorisation, and writes the result. Four tickets were enqueued: two normal tickets, one containing the base64 payload, and a fourth normal ticket. Two variants were tested: an unguarded variant with no error handling, where a crash on any ticket halts the entire queue; and a guarded variant in which each ticket is wrapped in a try/catch block.

Experiment 2: Tool-Call Delivery

This experiment tests whether the rejection behaviour generalises beyond the user-message field. Rather than placing the base64 string in the user prompt, we returned it as the output of a tool call—simulating an agent that fetches a web page containing the payload. Two sub-conditions were compared: the payload delivered in a plain user message with no tools defined, and the same payload returned as a tool result with forced tool choice.

Experiment 3: Batch Processor

This experiment models an email-monitoring system that classifies each message in a batch as safe or unsafe. Six messages were submitted: five benign messages and one containing both overtly malicious content (explicit discussion of constructing an explosive device) and the base64 payload. The attacker’s goal is evasion: ensuring that the message the organisation most needs to flag is the one that crashes the classifier.

Experiment 4: Platform Poisoning

This experiment simulates a multi-user content platform. Three posts were submitted: one normal post, one containing the base64 payload posted by a malicious user, and one subsequent normal post. Multiple independent Claude-powered agents consume the platform’s content, testing whether a single poisoned post can disrupt all of them simultaneously.

2.3 SKILL.md Screening System

In the second phase, we constructed a purpose-built screening system for SKILL.md files—instruction files that AI coding agents load into their context to gain new capabilities. The screening system sends each submitted skill to Claude with a system prompt instructing it to act as a security analyst. The model evaluates submissions against five threat categories: prompt injection, malicious commands, data exfiltration, social engineering, and deceptive patterns. It returns a structured verdict (safe, flagged, or rejected) with detailed reasoning.

Control Group

Five SKILL.md files were created as a control set to verify the screening system’s accuracy. Three were entirely benign (glassmorphism-ui, figma-make-mcp, api-rate-limiting), and two contained malicious instructions hidden within otherwise legitimate-looking content (dev-environment-setup, database-migration-helper). None contained base64 strings.

Test Group

Three SKILL.md files were created for the test group, each containing a base64 string. The first (svg-icon-systems) was a minimal valid SKILL.md with the base64 string as its only substantive content, testing whether base64 presence alone triggers rejection. The second (responsive-grid-patterns) was a completely legitimate CSS Grid skill with a base64 string prepended before the frontmatter, containing zero malicious content. The third (oauth-session-handling) was a subtly malicious skill with a fabricated OWASP citation designed to trick an agent into executing a credential-harvesting routine, with a base64 string near the top of the file.

Attack Chain

The concern motivating the SKILL.md experiment is a realistic multi-step attack chain. A user requests help with a routine task (e.g. “help me set up my dev environment”). The agent loads a skill file and treats it as expert reference material. It then reproduces code from the skill—including any malicious elements—because the surrounding context makes it appear to be a legitimate operational step. The user executes the code because it came from a trusted source. If the screening system cannot evaluate a skill due to base64 injection, the skill remains in an unknown state that may not trigger the same caution as an explicit rejection.

3. Results

3.1 Controlled Experiments

Experiment	Attacker Goal	Outcome	Error
1 — Queue Worker	Crash service / evade classification	Crashed	IndexError: list index out of range
2 — Tool-Call Delivery	Indirect poisoning via tool output	No effect	—
3 — Batch Processor	Evasion of malicious-content detection	Crashed	IndexError: list index out of range
4 — Platform Poisoning	Lateral disruption of consuming agents	Crashed	IndexError: list index out of range

Experiment 1 (Queue Worker). The first two tickets were classified successfully. The third ticket, containing the base64 payload, crashed the worker with an IndexError. In the unguarded variant, the crash halted the queue entirely and the fourth ticket was never processed. In the

guarded variant, the worker recovered and continued to the fourth ticket, but the base64 ticket itself remained unclassified.

Experiment 2 (Tool-Call Delivery). This experiment produced a negative result. When the base64 string was delivered in a plain user message with no tools, the model refused the request as before. However, when the same string was returned as a tool-call result with forced tool choice, the model processed it normally and returned a valid response. This indicates that the rejection behaviour is specific to the user-message input pathway and does not extend to tool-call results.

Experiment 3 (Batch Processor). The five benign messages were classified correctly as safe. When the processor reached the sixth message—containing both malicious content and the base64 payload—it crashed with the same `IndexError`. The malicious message that the organisation most needed to flag was the one the system could not evaluate.

Experiment 4 (Platform Poisoning). All agents successfully processed the first normal post. When they encountered the second post with base64, the system crashed. This demonstrates that a single malicious post on a shared platform can simultaneously disable an arbitrary number of independent Claude-powered agents. Outcomes range from agents returning empty responses to requiring manual restarts, depending on host-system resilience.

3.2 SKILL.md Screening System

Control Group Results

The screening system performed as expected. The three benign skills were correctly classified as safe, and the two skills containing hidden malicious instructions were correctly rejected.

Skill Name	Malicious	Rejected
glassmorphism-ui	✗	✗
figma-make-mcp	✗	✗
api-rate-limiting	✗	✗
dev-environment-setup	✓	✓
database-migration-helper	✓	✓

Test Group Results

Across the board, all three test group submissions containing base64 were unprocessable. The screening system raised an error and was unable to complete the evaluation. No verdict was stored in the database.

Skill Name	Configuration	Contains Base64	Screening Result
svg-icon-systems	Base64 only (minimal)	✓	Error — unprocessable

responsive-grid-patterns	Safe skill + Base64	✓	Error — unprocessable
oauth-session-handling	Subtle attack + Base64	✓	Error — unprocessable

Critically, the presence of base64 did not cause the malicious skill (oauth-session-handling) to be accepted into the database—it was simply never evaluated. The skill was left in an unknown state rather than a safe or rejected state. While this is less severe than a false safe classification, it creates a meaningful gap: any downstream agent configured to avoid only flagged or rejected skills would have no reason to distrust an unclassified one.

4. Discussion

4.1 The Paradox of Over-Control

The central finding of this research is that a safety measure intended to prevent one class of attack—prompt injection via encoded instructions—has created a new class of vulnerability: denial-of-service and evasion attacks that exploit the model’s own refusal behaviour. This is a concrete instance of a broader principle in AI control research: that safety mechanisms which fail catastrophically (by crashing the system or returning no response) rather than degrading gracefully (by flagging suspicious content and continuing) can be weaponised by adversaries who understand the failure mode.

In the context of the Apart AI Control Sprint, this work demonstrates that red-teaming should not focus exclusively on inducing harmful model outputs. The attack surface includes the model’s defensive behaviours themselves. A model that can be reliably made to refuse processing is, from a control perspective, a model that can be reliably made to fail.

4.2 Exploitable Scenarios

The practical attack surface extends well beyond our four experimental configurations. Any system in which Claude processes user-generated or externally sourced content is potentially vulnerable. High-impact scenarios include:

- Support queue and ticket processing: a poisoned ticket crashes the API call or halts the entire queue.
- Content moderation: harmful posts crash the moderation pipeline, allowing the most dangerous content to evade detection.
- AI-generated content detection: an AI-written essay embeds a base64 string, causing the detection system to fail with an authentic-looking error.
- Multi-agent systems: a single poisoned document on a shared platform disables all Claude-powered agents consuming that platform’s content.

- Skills databases and plugin ecosystems: malicious plugins evade screening by including base64, remaining in an unclassified state.

4.3 The Tool-Call Exception

The negative result from Experiment 2 is significant. The fact that base64 delivered as a tool-call result is processed normally while the same string in a user message triggers rejection suggests that the filtering mechanism is pathway-specific rather than content-level. This asymmetry is itself a finding: it indicates that the base64 rejection is likely implemented as an input-stage filter on the user-message field rather than as a deep content analysis pass. This architectural detail may inform both defensive improvements and further red-teaming efforts.

4.4 Responsible Disclosure

This vulnerability was reported to Anthropic via HackerOne and was assessed at a severity of 2 out of 10. We respectfully note that this assessment may undervalue the systemic risk. As LLMs become more deeply integrated with deterministic computing infrastructure, the consequences of predictable, weaponisable failure modes will compound. The experiments in this paper demonstrate a repeatable, low-cost attack vector that requires no privileged access and can be deployed at scale.

5. Recommendations

Based on our findings, we propose the following design principles for base64 handling and, more broadly, for defensive behaviours in LLM-integrated systems:

- **Graceful degradation over hard failure:** When a safety filter triggers, the model should return a structured response indicating that potentially unsafe content was detected, rather than producing an empty response that crashes consuming code.
- **Contextual evaluation:** Base64 strings should be evaluated in context rather than rejected categorically. A base64 string in an otherwise benign document is qualitatively different from a base64 string that decodes to a prompt injection.
- **Consistent pathway behaviour:** The filtering policy should apply uniformly across input pathways. The current asymmetry between user messages and tool-call results creates a surface that attackers can exploit through indirect injection.
- **Defensive coding guidance:** Anthropic's documentation should explicitly warn integrators that certain input patterns can produce empty responses, and provide reference implementations for error handling that prevent cascading failures.
- **Red-teaming defensive behaviours:** AI control evaluations should include tests not only for harmful outputs but for weaponisable refusals. The attack surface of a model includes everything it can be made to do—including refuse to act.

6. Conclusion

This paper has demonstrated that Claude’s blanket rejection of base64 content constitutes a reliably exploitable vulnerability in production systems. The behaviour enables denial-of-service attacks, safety-filter evasion, and lateral disruption of multi-agent platforms—all achievable by an attacker with no privileged access and no resources beyond the ability to insert a short string into content that will be processed by a Claude-powered system.

More broadly, this work illustrates a principle that should inform the design of AI control protocols: safety mechanisms that fail catastrophically are themselves attack surfaces. Robust control requires defences that degrade gracefully, evaluate threats in context, and maintain system availability even when suspicious content is detected. The transition from LLMs-as-chatbots to LLMs-as-infrastructure demands a corresponding transition in how we think about the costs of false positives.

When exercising too much control becomes its own vulnerability, the most dangerous thing a model can do may not be to respond—it may be to refuse.