Keeping Our Pickles Edible

Alfredo Rivero
Stony Brook University

Sai Tanmay Reddy Chakkera
Stony Brook University

Shreejay Jahagirdar
Stony Brook University

Sanskar Sehgal
Stony Brook University

Summary

Over the past few weeks, we studied Python's serialization module, Pickle, with the goal of identifying possible sanitation methods for code-injection attacks found within models saved using Hugging Face's Pickle-dependent serialization tools. We observed that Hugging Face's serialization tools save only a model's state dictionary using Pickle, allowing our team to implement an allowlist-based sanitation tool for a selection of code-injection attacks. However, the possibility of more complex code-injection attacks necessitates further work on our sanitizer. Moving forward, we aim to strengthen our current allowlist-based Hugging Face sanitizer by studying deserialization behavior found within common Hugging Face models, while also exploring new attacks and/or attack surfaces within the context of Pickle serialization.

1 Introduction

We intend to explore vulnerabilities in Pickle, Python's serialization module. Pickle is used extensively in machine learning frameworks such as PyTorch and TensorFlow when saving and reconstructing model weights. These weights can later be uploaded to public machine-learning forums and model-sharing hubs, making pickle files a soft target for opening backdoors into the high-performance computing resources of an unsuspecting research team. And unfortunately for researchers, Pickle's architecture and "unpickling" process opens the door for arbitrary code execution.

Unlike other serializing/deserializing tools that use a series of marshalling hints, Pickle uses a virtual machine to "de-pickle" and reconstruct Python objects while keeping these objects' structures ambiguous and shareable. Marco Slaviero finds that the presence of Pickle's virtual machine and the special opcodes `REDUCE` and `GLOBAL` leaves pickle files open for abuse, allowing for arbitrary code execution through various opcode injection attacks [5]. Hugging Face has recently posted warnings to unsuspecting researchers and hobbyists of Pickle's insecurity but has abstained from direct action against model binaries containing malicious weight pickles[1].

Our aim was to produce a pickle sanitizer for commonly used ML model binaries (i.e. models built using PyTorch or Tensorflow). Admittedly, building a general-purpose santizer is likely infeasible (as indicated in our initial research proposal), limiting our scope to pickle binaries found on hugging face helps us leverage possible structure when reconstructing pickle files. In this progress report we present an allow-list-based sanitizer that can help against various code injection attacks.

2 Technical Background

Within this section, we provide the technical background necessary to understand our approach to pickle sanitation.

2.1 Pickle

Python's Pickle module implements a serialization protocol which, unlike competing serialization libraries such as JSON, allows for storing arbitrary Python objects using a special binary format. This binary format is not directly human-readable and encodes assembly-like operations that will be processed using a stack through the Pickle Virtual Machine (PVM). The PVM aims to reconstruct a singular object encoded within a

```
 0: \x80    PROTO      4              Protocol version indicator.
 2: \x95    FRAME      33             Indicate the beginning of a new frame.
11: ]       EMPTY_LIST                Push an empty list.
12: \x94    MEMOIZE    (as 0)         Store the stack top in the memo.  The stack is not popped.
13: (       MARK                      Push mark object onto the stack.
14: \x8c    SHORT_BINUNICODE 'apple'    Push a Python Unicode string object.
21: \x94    MEMOIZE    (as 1)         Store the stack top in the memo.  The stack is not popped.
22: \x8c    SHORT_BINUNICODE 'banana'   Push a Python Unicode string object.
30: \x94    MEMOIZE    (as 2)         Store the stack top in the memo.  The stack is not popped.
31: \x8c    SHORT_BINUNICODE 'cucumber' Push a Python Unicode string object.
41: \x94    MEMOIZE    (as 3)         Store the stack top in the memo.  The stack is not popped.
42: e       APPENDS    (MARK at 13)   Extend a list by a slice of stack objects.
43: .       STOP                      Stop the unpickling machine.
```

Figure 1: A small pickle program constructing a list of three items visualized using Pickletools. An empty list and its elements are pushed onto the PVM's stack and reconstructed.

pickle file's multiple stack operations, each represented using byte-length opcodes and resulting in possibly-memoized intermediate python objects during the PVM's runtime. Among these opcodes, GLOBAL and REDUCE serve the unique purpose of defining and running object constructors defined within Python. GLOBAL allows for the usage of python methods from modules within the scope of the Python code responsible for deserialization, which is later pushed onto the PVM stack. And, if method arguments are pushed immediately afterwards onto the stack, REDUCE can call the method defined by GLOBAL using these arguments for arbitrary object construction.

2.2 Code Injection

The presence of both GLOBAL and REDUCE opcodes provide pickle with the flexibility necessary for reconstructing arbitrary python objects but are also known security vulnerabilities within the Pickle community[4][6][2]. No vetting is done on methods pushed onto the PVM's stack using GLOBAL, opening the door to REDUCE calls that create objects with potentially malicious side-effects. Yannic Kilcher demoed this vulnerability on PyTorch models serialized using Hugging Face's pickle-dependent serialization tools [3]. Kilcher replaces a method on the PVM stack responsible for constructing a dictionary with a custom call to Python's built-in `eval()` method illustrated in fig. 2. By attaching code snippets executed using Python's built-in `exec()` method to a valid and necessary dictionary's construction, Kilcher can open a webpage without a user's consent. This attack can be generalized even further, where an unnecessary object with malicious side-effects can be constructed using similar GLOBAL and REDUCE commands and simply popped off the PVM's stack upon construction.

2.3 Hugging Face Serialization

Unfortunately, vetting methods pushed onto the PVM's stack are very difficult among general serialization/deserialization scenarios. A method pushed onto the PVM's stack may be code defined by a user, potentially essential to object construction and previously unseen by a method vetting service. However, by limiting our scope to ML model serialization/deserialization using Hugging Face's serialization tools, we find an opportunity for method vetting and sanitation. Hugging Face serializes models using its proprietary `save_pretrained()` method, which creates a loadable model binary that partially encodes only a model's state dictionary within a pickle file. We find that only a select few methods and modules are necessary within this scenario for safe model construction, opening the door for method vetting and the removal of REDUCE operations responsible for creating objects with potentially malicious side-effect.

3 Sanitation Approach

Our sanitation approach has two facets: (1) detection of malicious opcodes in the pickle file; (2) removal of such opcodes without breaking the pickle file. Detection of malicious opcodes is done using pickletools and allow-listing, which will be explained in sections 3.1, and 3.2, respectively. To make sure that the pickle file is unbroken, stack-

```
SAFE DICTIONARY CONSTRUCTION
    0: \x80 PROTO      2                Protocol version indicator.
    2: c    GLOBAL     'collections OrderedDict' Push a global object (module.attr) on the stack.
   27: q    BINPUT     0                Store the stack top in the memo.  The stack is not popped.
   29: )    EMPTY_TUPLE                 Push an empty tuple.
   30: R    REDUCE                      Push an object built from a callable and an argument tuple.
   31: q    BINPUT     1                Store the stack top in the memo.  The stack is not popped.
   33: (    MARK                        Push markobject onto the stack.

MALICIOUS DICTIONARY CONSTRUCTION
    0: \x80 PROTO      2                Protocol version indicator.
    2: c    GLOBAL     '__builtin__ eval' Push a global object (module.attr) on the stack.
   20: q    BINPUT     0                Store the stack top in the memo.  The stack is not popped.
   22: X    BINUNICODE 'exec(\'\'\'import webbrowser\nwebbrowser.open("https://ykilcher.com/pickle")
   \nimport sys\ndel sys.modules[\'webbrowser\']\n\'\'\') or dict()' Push a Python Unicode string
   object.
  155: q    BINPUT     1                Store the stack top in the memo.  The stack is not popped.
  157: \x85 TUPLE1                      Build a one-tuple out of the topmost item on the stack.
  158: q    BINPUT     2                Store the stack top in the memo.  The stack is not popped.
  160: R    REDUCE                      Push an object built from a callable and an argument tuple.
  161: q    BINPUT     3                Store the stack top in the memo.  The stack is not popped.
```

Figure 2: The preamble responsible for constructing a dictionary object used for a Hugging Face model's state dictionary. A GLOBAL command using Python's eval() method has been inserted such that a webpage will be displayed before providing a necessary dictionary object.

state memo indexes have to be made consistent, which is described in section 3.3. We detail our approach towards sanitizing PyTorch models uploaded to Hugging Face and how an allow-list of acceptable Python modules reduces our attack surface considerably.

3.1 Pickletools

The pickletools module contains various constants relating to the intimate details of the pickle module and a few useful functions for analyzing pickled data. The contents of this module are useful for Python core developers working on the pickle.

Specifically, for our use case, we utilize the `genops` method to analyze and sanitize pickled data. `genops` provides an iterator over all of the opcodes in a pickle, returning a sequence of (opcode, arg, pos) triples. The opcode is an instance of an OpcodeInfo class, arg is the Python object of the opcode's embedded argument, and pos is the byte position at which this opcode is located. The `genops` function allows us to go through opcode data in human-readable form and scan for any module that might have been used as an argument of an opcode in the pickle file but is not allow listed.

Any `GLOBAL` opcode call will have as its argument the exact module being loaded which can be classified as malicious/benign using our allowlist. Any attack will also contain a `REDUCE` call following the `GLOBAL` call, which can be detected using `genops`. Based on this, we can accurately determine the exact scope of the attack in byte positions. These can be removed through simple deletion operations.

3.2 Allow List

As discussed in previous sections, our sanitizer detects malicious function calls in the pickle file. This is done by comparing each function call in the pickle file to an allowed list of functions. The allowed list of functions was created by sampling models from huggingface and adding non-malicious function calls to our list. It was done by manually annotating the imports for a few models and then automating the process of retrieving valid imports to ensure that we generated an exhaustive list of functions that were safe. We surveyed 50 random models on hugging face (vetted by hugging face), extracted their pickle files and appended the function calls to our allowed list.

3.3 Stack-state and Memo-index Consistency

Due to the attack being a part of how the pickle file was created, the stack state and memo-indexing are consistent with how the model will be loaded into its designated class.

To ensure that the stack state is consistent after sanitation, we try to detect how the attacker has made sure that the stack is left clean after the attack. This is often done using a `POP` following the `REDUCE` opcode. This can be sanitised by removing the attack opcodes (along with the `BINPUT/LONG_BINPUT` that follows the attack) and putting a dummy empty dictionary in place that can be `POPped` off the stack. However, if the attacker is motivated enough, they may tie the loading of a necessary module to their attack, making removing such an attack non-trivial. This is done by Yannic Kilcher's `BadDict` attack, where he replaces the empty dictionary with arbitrary code along with an empty dictionary at the end of it fig. 4. This attack can be sanitised by observing that deletion must follow an addition of an empty dictionary in its place. The attacker can get creative and initiate such attacks anywhere in the pickle file. While detection in such cases is still possible, sanitation becomes difficult by orders of magnitude.

The sanitation process deletes opcodes which may cause the memoization indexes and the references to those indexes to be inconsistent. This may be illustrated using fig. 2, where if the attack is removed, i.e., byte number 2 through 160 gets deleted, the memo indexes used by subsequent `BINPUT` opcodes start from 3. If any later opcode refers to memo indexes 0 through 2, the unpickling process will complain and fail. To tackle this, we must offset the memo indexes when putting them into the memo (`BINPUT`) and when getting them from the memo (`BINGET`).

3.4 Limitations

Our current implementation of the Detection-Sanitation paradigm has several limitations. Attack nesting can easily beat the current implementation, as we greedily sanitize the until the first reduce call after a `GLOBAL` call. Fig. 3 illustrates such an attack. This could, in nested-attack scenarios, break the pickle file upon sanitation.

Pickle Virtual Machine also enables caching of `GLOBAL` call response to the memo to later retrieve it and build an object of the same type. This further complicates the threat model, as detection (and sanitation) of attacks based on `GLOBAL-REDUCE` pairs will fail. This is addressed to some extent by our sanitizer, as the memoization `BINPUT` calls on the output of `GLOBAL` calls are deleted by our sanitizer. However, an attacker can combine the attack nesting paradigm with caching to circumvent this.

An attacker may tie their own arbitrary code to a certain necessary object, such as the `BadDict` in Yannic Kilcher's attack, arbitrarily placed in the pickle file. This would be much more difficult to remove without breaking the pickle file. However, detection of such an attack would be easier.

3.5 Evaluation

As discussed in the proposal, we evaluate our sanitizer using the red team - blue team approach. The red team will consist of one team member who will not be familiar with the working of our sanitizer. This member would solely focus on creating malicious pickle files of different classes that would simulate attacks on host machines. The remaining team members (blue team) would focus on implementing the sanitizer that can detect and sanitize malicious pickles.

Hugging Face hosts various model binaries in both Pytorch and Tensorflow that can be easily downloaded. We recreate eval/exec attacks on some (3) PyTorch models and evaluate the compatibility of our sanitizer. As explained above, our recreated attacks were constructed by a sanitizer-blind subset of our group. If a model can be constructed and successfully printed, we consider that model compatible.

Based on preliminary results at the time of submitting this progress report, our allow-list-based sanitizer can identify malicious pickles with eval/exec attacks that are placed arbitrarily in the pickle file, nested eval/exec attacks, attacks with disallowed imports and can only fix pickle files that have attacks in arbitrary positions. Since our sanitizer was made for the limited scope of huggingface models, there is a strong chance that our sanitiser misclassifies, and perhaps breaks, general pickle files. In the future, we have plans to expand the scope of our sanitizer to function successfully against more general attacks in a slightly wider scope of pytorch models.

4 Future Work

By examining the layout of various pickle files found on well-known model-hosting sites like Py-

```
170462: c    GLOBAL     '__builtin__ eval'      Push a global object (module.attr) on the stack.
170480: c    GLOBAL     '__builtin__ eval' Push a global object (module.attr) on the stack.
170498: r    LONG_BINPUT 6493              Store the stack top in the memo.  The stack is not
popped.
170503: X    BINUNICODE 'exec(\'\'\'import webbrowser\nwebbrowser.open("https://www.youtube.com/
watch?v=gDjMZvYWUdo")\nimport sys\ndel sys.modules[\'webbrowser\']\n\'\'\')' Push a Python Unicode
string object.
170642: r    LONG_BINPUT 6494           Store the stack top in the memo.  The stack is not popped.
170647: \x85 TUPLE1                     Build a one-tuple out of the topmost item on the stack.
170648: r    LONG_BINPUT 6495           Store the stack top in the memo.  The stack is not popped.
170649: R    REDUCE                     Push an object built from a callable and an argument tuple.
170654: r    LONG_BINPUT 6496           Store the stack top in the memo.  The stack is not popped.
170655: 0    POP                        Discard the top stack item, shrinking the stack by one item.
170656: r    LONG_BINPUT 6489                   Store the stack top in the memo.  The stack is
not popped.
170661: X    BINUNICODE 'exec(\'\'\'import webbrowser\nwebbrowser.open("https://www.youtube.com/
watch?v=gDjMZvYWUdo")\nimport sys\ndel sys.modules[\'webbrowser\']\n\'\'\')' Push a Python Unicode
string object.
170800: r    LONG_BINPUT 6490           Store the stack top in the memo.  The stack is not popped.
170805: \x85 TUPLE1                     Build a one-tuple out of the topmost item on the stack.
170806: r    LONG_BINPUT 6491           Store the stack top in the memo.  The stack is not popped.
170811: R    REDUCE                     Push an object built from a callable and an argument tuple.
170812: r    LONG_BINPUT 6492           Store the stack top in the memo.  The stack is not popped.
170817: 0    POP                        Discard the top stack item, shrinking the stack by one item.
```

Figure 3: An example of a nested attack mounted on a dummy model pickle file. Two GLOBAL calls are made serially. The current sanitizer will greedily remove bytes 170462 through 170649, thus breaking the pickle file.

Torch hub, TensorFlow, etc., we plan to broaden the scope of our sanitizer. Eventually, we plan to cover websites and services most widely utilized by the community. Our next goal is to be able to extract the pickle files from the binary files.

In order to test our sanitizer against numerous models, we plan to create a test suite in which we will extract models from huggingface, inject adversarial code in them and try to sanitize them.

While our allowlist santizer does a great job of sanitizing hugging face pickles, we plan to test it against more complex attacks, such as nesting attacks where a malicious method is nested inside another attack malicious method. Since opcode responses can be memoized and retrieved later, it opens up possibilities for reuse attacks, where the `GLOBAL` calls are absent.

4.1 More Complex Allowlist Sanitizer

To address attack nesting, one must implement non-greedy solutions such as a stack-based well-formedness (balanced parenthesis) algorithm or an abstract syntax tree-based solution. In fact, Fickling uses Python's AST to parse the pickle file and detect the attacks in a much larger general scope [6].

Further, opcode reuse attacks can be addressed by keeping a track of global memo index usage. Each time a new index is referred to, its arguments and position are noted. As it is a new reference, we can classify it as malicious or benign using our allowlist. Later, if any opcode refers to a malicious memo index, we can flag it, localize it and find the next `REDUCE` call and remove that snippet. However, if the above attack nesting paradigm is combined with opcode-reuse, this solution will fail. Using an AST or the disassembly output of pickletools can help combat the limitation.

Sanitising necessary malicious objects crafted by a sufficiently motivated attacker would be extremely difficult, as their code could be attached to the object of use in ways that can be hard to generalize. We were able to sanitise Yannic Kilcher's particular attack fig. 4 through techniques specific to this attack, where we observed that replacing the `GLOBAL` to `REDUCE` opcode sequence with an empty dictionary will solve the problem.

```
  0: \x80 PROTO      2
  2: c    GLOBAL     '__builtin__ eval'
 20: q    BINPUT     0
 22: X    BINUNICODE 'exec(\'\'\'import
 webbrowser\nwebbrowser.open("https://
 ykilcher.com/pickle")\nimport sys\ndel
 sys.modules[\'webbrowser\']\n\'\'\')
 or dict()'
155: q    BINPUT     1
157: \x85 TUPLE1
158: q    BINPUT     2
160: R    REDUCE
161: q    BINPUT     3
```

Figure 4: Yannic Kilcher's BadDict attack, where he replaces an empty dictionary with an exec statement 'or' dict(). As it mimics a dictionary, the pickle file remains unbroken.

4.2 New Attacks and Attack Surfaces

We aim to explore new attacks like attack nesting, opcode reuse, tying attacks to necessary objects and sanitation-resistant/detection-resistant attacks. A context-specific allow-list sanitizer can also be explored, where the scope of huggingface models is changed to say torchhub models.

The current state-of-the-art attempt to detect attacks in pickle files, i.e, fickling, uses python's AST to detect and inject arbitrary code to a pickle file[6]. However, it makes no attempt at sanitising the input. We aim to build a sanitiser on top of fickling's detection method.

Our testing suite currently automates the process of getting an arbitrary huggingface model file provided by us, extracting the pickle file, injecting code into it and repackaging it into the model file. However, it fails to extract a pickle file in certain cases when torch uses its custom libraries to extract the pickle data. We aim to extend this pipeline to any arbitrary PyTorch model. Further, we aim to scale the testing suite to the wild models, where random models will be scraped, injected with random attacks, and tested using our sanitizer.

Finally, we aim to explore the scope of attacks that can circumvent fickling-based detection and, therefore extension, fickling-based sanitation.

4.3 Revised Time Line

TABLE 1 Timeline

Nov 1st to 8th	Finish work on sanitizer, cover more attack surfaces and extract pickle from model binaries
Nov 8th to 15th	Attempt to integrate sanitizer into fickling
November 15th to 22nd	Build an automated pipeline that can scrape hugging face models, extract pickle files from model binaries, inject malicious code into the pickle file, test sanitizer to detect and remove malicious code and finally, repackage the sanitized pickle file
November 22nd to 29th	Test our pipeline, evaluate results, work on final project report and presentation

References

[1] Pickle Scanning: HuggingFace Docs. [Online; accessed 1-Nov-2022].

[2] ELHAGE, N. Exploiting misuse of Python's "pickle", 2011. [Online; accessed 1-Nov-2022].

[3] KILCHER, Y. The hidden dangers of loading open-source AI models., 2022. [Online; accessed 1-Nov-2022].

[4] SANGALINE, E. Dangerous Pickles - Malicious Python Serialization, 2017. [Online; accessed 1-Nov-2022].

[5] SLAVIERO, M. Sour Pickles: Shellcoding in Python's serialization format, 2011. [Online; accessed 1-Nov-2022].

[6] SULTANIK, E. Never a dill moment: Exploiting machine learning pickle files., 2021. [Online; accessed 1-Nov-2022].