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

We studied Python's serialization module, Pickle, with the goal of identifying possible sanitation methods for code-injection attacks found within models saved using PyTorch'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. We generalize this tool for all PyTorch models saved using a `state_dict()`. While building our sanitizer, we also identified a difficult-to-sanitize and separate selection of code-injection attacks we refer to as interwoven exploits. Finally, we propose our allowlist sanitizer as a security tool adaptable to other pickle files, most notably models built using Sci-Kit Learn.

1 Introduction

Pickle, Python's serialization module, is used extensively in machine learning frameworks such as PyTorch 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 host machines and potentially extensive computing resources. And unfortunately for researchers, Pickle's architecture and deserialization process is susceptible to arbitrary code execution.

Unlike other serialization 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 researchers and hobbyists of Pickle's insecurity but has abstained from direct action against model binaries containing malicious pickle files [1]. We believe the difficulty of vetting all models' safety has discouraged direct action, as model diversity and using different model serialization tools are common within machine learning. Indirect protections against malicious binaries, including hash-based model authentication schemes, could be considered but would not protect against malicious model authors or malicious third-party model distributors.

Thus, our aim is to limit our supported model scope and produce a pickle sanitizer for, initially, a single kind of ML model binary: PyTorch models serialized using `state_dict()` proprietary serialization tools. By doing so, we can examine Pickle's serialization behaviour within a restricted environment, where only model weights are stored within a model's pickled data. This restricted environment allowed us to identify operations relevant only to weight reconstruction and employ an allowlist of acceptable python code used during reconstruction. Ultimately, we find that our allowlist approach has limitations imposed by a particular class of attacks that mix malicious and necessary reconstruction operations, but we also find that our approach is flexible enough to apply to other models and pickle files.

2 Technical Background

This section provides the technical background necessary to understand our approach to pickle

```
 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.

sanitation within a restricted PyTorch serialization environment.

2.1 Pickle

Python's Pickle module implements a serialization protocol that, unlike competing serialization libraries such as JSON, allows 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 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.

The pickle virtual machine consists of a stack that functions as a scratch table, and a memo, that functions as a cache, to store intermediate results during the deserialization operation. `BINPUT/LONG_BINPUT` with a specific index as argument are used to store the top of the stack into the memo at the index, in pickle protocol 3 or below. `MEMOIZE` is used to perform the same function, however without any index, protocol 4 onwards. `BINGET` is used to load specific indexes from the memo to the top of the stack. `BINUNICODE` is used to load a unicode string onto the top of the stack. `GLOBAL` is used to load foreign modules and `REDUCE` is used to construct the object called by `GLOBAL`.

Among these opcodes and pickle protocols 1-3, `GLOBAL` and `REDUCE` serve the unique purpose of defining and running object constructors within Python. `GLOBAL` allows for the usage of python methods from modules within the scope of the Python code responsible for deserialization. These methods are pushed onto the PVM's 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. Pickle protocol 4 onwards, `GLOBAL` is replaced by two `BINUNICODE`s indicating the name of the module being loaded, and a `STACK_GLOBAL` opcode.

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.

```
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.

2.3 PyTorch Serialization

Unfortunately, vetting methods pushed onto the PVM's stack is very difficult among many Pickle serialization/deserialization scenarios. It is not uncommon for Pickle files to store GLOBAL and REDUCE operations that call object constructors during deserialization. These constructors are potentially unique, previously unseen, and essential during deserialization. However, by limiting our scope to ML models serialized using PyTorch's `state_dict()` serialization tools, we find an opportunity for method vetting and sanitation. As mentioned in our progress report, Hugging Face are particularly tractable due to its serializing models using its `save_pretrained()` method, which in turn saves the model as a state dictionary using `torch.save()`. `torch.save()` creates a loadable model binary that serializes only a model's state dictionary (in the form of a builtin dict or an OrderedDict) within a pickle file. We find that only a select few methods and modules (such as tensor base classes, floatStorage, ordered dictionary etc.) 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 Possible Code Injection

We identified two possible directions for code injection within Pickle files. First, would be relying on executing Python methods defined within the scope of deserialization. Under this direction, methods from already-imported modules would be passed into the PVM using the `GLOBAL` opcode and executed with a `REDUCE` call. One limitation of this approach is an attacker's inability to execute any code outside of the deserializer's scope. This limits possible attacks and leaves a malicious party guessing what python code is within reach during serialization. However, one module is always within reach for all Python code executed: __builtins__. Our second direction for code injection relies on exclusively pushing methods from this module, specifically eval() and exec(), onto the PVM's stack for near arbitrary code execution. We find the greatest flexibility for code injection starting from this second direction.

3.1 __builtins__'s eval and exec

Python's __builtins__ module provides direct access to all "built-in" identifiers of Python, including essential operations such as Python's range() and open() calls used while looping and for file in-

put/output. Among these operations, eval() and exec() are included. Both serve similar purposes of code execution using string-bound code snippets or singular expressions, with exec() in particular allowing for unrestricted code execution within the scope of deserialization. Using exec(), a malicious attacker can circumvent the deserializing code's scope, importing any modules necessary for an attack and even the ability to cover their tracks by deleting any imported/defined modules/variables.

3.2 Attacks

All attacks developed inject and execute methods within a deserializer's scope. Due to Pickle files' lack of human readability, attacks are injected using direct byte manipulation. Specifically, malicious `GLOBAL` and `REDUCE` reduce operations are introduced in a variety of configurations. As long as

3.2.1 Sequential Execution

This is the simplest attack we developed, and can make use of the PVM's memo for repeated and more complex code injection. This attack and its memoized variant can be describe as follows:

0. If implementing a memoized variant of this attack, insert your desired malicious method into the PVM's memo using BINPUT or LONG_BINPUT.
1. Push the desired malicious method onto the PVM's stack using GLOBAL. If implementing a memoized variant, push the desired malicious method by retrieving it from the PVM's memo using BINGET or LONG_BINGET instead.
2. Push the malicious method's arguments onto the PVM stack as unicode strings using BINUNICODE.
3. Create a tuple of all arguments using TUPLE opcodes including TUPLE1, TUPLE2, etc.
4. Make a REDUCE call using our malicious method and argument tuple. This will create an object on the stack.
5. Pop the REDUCE call's created object off the stack using a POP call.

This attack type can be safely inserted into any pickle file as long as the PVM's memo is unaffected (i.e. nothing within these steps is memoized). If something is memoized, successive calls to BINGET and LONG_BINGET must have their memo indices increased by the number of items memoized throughout the attack.

3.2.2 Nested Execution

This attack is a variant of the previous attack designed to obfuscate sanitation through recursive code execution. After any previous attack steps, recur back to step 0. Execute these new steps and optionally recur as much as necessary. This attack functions due to the previous attack's usage of only the topmost stack elements, leaving any previous stack elements on the stack and unmodified.

3.2.3 Interwoven Exploits

- Introduce dependencies into attack. - Can be done

3.2.4 "Bad Dict" Exploit

- This is really similar to sequential execution, but also an example of an interwoven exploit. - Do same, but eval() exec() - Appears at the start of the program, replaces dict with malicious equivalent

3.3 Sci-Kit Learn and Pickle Protocol 4

4 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 4.1, and 4.2, respectively. We describe our allow-list-based sanitation method for PyTorch `state_dict()` models in more detail in section 4.3.

4.1 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`. In protocol 4 and above, the `GLOBAL` call is replaced by a `STACK_GLOBAL` call and preceded by two `BINUNICODE`s, indicating the exact module being loaded. This will also be followed by a `REDUCE` call. This scope of `GLOBAL`-like to `REDUCE` opcodes can be removed through simple deletion operations.

Thus, using `genops`, we can accurately determine the exact scope of the attack in byte positions.

4.2 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.

4.3 Sanitizer

The process of sanitation begins with an input binary file. PyTorch uses multiple methods to package models into binaries, a few of which are zip-like, tar-like and stack-of-pickle files. Pickle files are extracted from their binaries based on the specific method used to build them. Next, pickle files are read, and the protocol number for the pickle file is detected. This is followed by a detection run on the pickle file and attack scope resolution on it. The attack scope resolution considers the `GLOBAL` opcode reuse using the memo and nesting of attacks. This resolved scope is deleted from the pickle file and replaced by an empty dictionary based on the need. However, a simple deletion of the attack scope will break the file.

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/MEMOIZE` that follows the attack) and putting a dummy empty dictionary in place that can be `POP`ped 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. 3. 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`). In protocol 4 and above, `BINPUT/LONG_BINPUT` gets replaced by `MEMOIZE`, for which index of the memo location to write is the number of elements currently present in the memo. This saves us the step of resetting the `BINPUT/LONG_BINPUT` arguments. However, `BINGET` are called similar to protocol 2, and their arguments need to be reset based on offsets.

Following these sanitation steps, a new file is created and written into. The pickle files are then put together again to reconstruct the binary file, which can be utilised as before.

4.4 Limitations

Our current sanitizer assumes all opcodes between a malicious `GLOBAL` and a malicious `REDUCE` call are infected. However, an adversary might include some code necessary for the execution of the file (secure and essential code) somewhere in between the scope of the attack. Passing this through our sanitizer will result in the removal of the entire scope (including the safe and essential parts), which will break the file during its execution. This can be tackled to some extent by distinguishing between attack opcodes and safe opcodes using symbolic execution.

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, due to the indistinguishability of attack code and useful code. However, detection of such an attack would be easier.

The latest major change to the protocol for pickling files is protocol - 4. Due to pickle's legacy protocol support, an adversary may inject an attack into a protocol - 4 pickle files using protocol - 2 opcodes, which our detector would be able to detect and flag. But our sanitizer cannot sanitize such files. This "mixing" of protocols by an adversary can be resolved by adding provisions in our tool for such attacks.

Due to the breakdown of `GLOBAL` into multiple opcodes in the form of `BINUNICODE`s and `STACK_GLOBAL` in protocol 4, new limitations to our method of detection and sanitation crop up. The attacker can be arbitrarily creative in the positioning of the two `BINUINCODE`s and `STACK_GLOBAL` in the pickle file such that the stack remains unchanged. This makes it difficult to detect the exact module being loaded, and sanitation even harder. This can be resolved to some extent by the above-mentioned method of symbolic execution to tell apart attack opcodes from original opcodes.

4.5 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 (8) 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 the above formulation, our allow-list-based sanitizer can identify malicious pickles with eval/exec attacks that are placed arbitrarily in the pickle file (if in protocol 2, else limited to unbound versions in protocol 4), nested eval/exec attacks, attacks with disallowed imports and can only fix pickle files that have attacks in arbitrary positions(without any kind of binding).

Further, in order to test our methods' versatile applicability, we construct an allowlist for sci-kit models and detect/sanitize them accordingly. We find that while sci-kit may have a longer allowlist due to not having a `state_dict()`-like method, it is equally detectable and sanitizable.

5 Future Work

We could explore a static analysis approach to distinguishing between attack opcodes and original opcodes, specifically symbolic execution. In fact, Fickling [6] uses a symbolic execution-based static analyser to detect attacks in a very broad setting. Adding an allowlist functionality to Fickling can improve our limitations in detection to a certain extent.

Additional provisions can be made in our tool to account for legacy support in pickle, in that we could add provisions to sanitize attacks of protocol-2 in pickle files of protocol-4.

Additionally, one can aim to test the existence of attacks in in-the-wild models, where random PyTorch (`state_dict()`) models will be scraped from the web and tested using our detector-sanitizer. This direction could help us in detecting new attacks.

6 Conclusion

We present our detector-sanitizer tool for PyTorch models saved using `state_dict()` using an allowlist-based approach. We extend this approach to a different scope of sci-kit learn models. Further, we present and describe tools to inject

```
  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 3: 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.

attacks into PyTorch model binary files and pickle files. This is followed by our limitations and ideas on resolving them to the extent possible.

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].