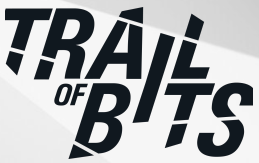

YOLOv7

Threat Model and Code Review

October 31, 2023

Prepared by: **Alvin Crighton, Anusha Ghosh, Heidy Khlaaf, Jim Miller, Kurt Willis, Maciej Domanski, Spencer Michaels, Suha Hussain, and William Woodruff**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Trail of Bits performed a lightweight threat model and secure code review of YOLOv7. YOLO, short for "You Only Look Once," is a popular model in computer vision used for real-time object detection. YOLO has gained popularity because it achieves high accuracy despite being only a single neural network that requires only a single evaluation. This makes YOLO particularly strong for applications that require detecting objects in real-time, such as autonomous vehicles.

A team of two consultants conducted the lightweight threat modeling exercise for YOLOv7 from May 30, 2023 to June 2, 2023, for a total of two engineer-weeks of effort. After the threat modeling exercise was completed, the team shared the results with the entire Trail of Bits team. Shortly after, a team of two different consultants performed a targeted secure code review of the YOLOv7 codebase from June 7, 2023 to June 16, 2023, for a total of two engineer-weeks of effort.

Our testing efforts focused on threat scenarios identified during the threat modeling exercise, such as the potential compromise of datasets used by YOLOv7. With full access to the YOLOv7 source code, original YOLO papers, and the YOLO documentation, we performed static testing of the codebase, using automated and manual processes.

Observations and Impact

As currently written, the YOLOv7 codebase is not suitable for security-critical applications or applications that require high availability, such as autonomous vehicles. We reach this conclusion for two main reasons. First, the codebase is not written or designed defensively. User and external data inputs are poorly validated and sanitized (TOB-YOLO-9). If an attacker is able to control or manipulate various inputs to the system, such as model files, data files, or configuration files, they could perform a denial-of-service attack with low effort (TOB-YOLO-8, TOB-YOLO-9, and TOB-YOLO-12). Other, more severe threats, such as arbitrary code execution, are also possible (TOB-YOLO-1, TOB-YOLO-2, TOB-YOLO-3, TOB-YOLO-4). Additionally, there are no unit tests or any testing framework in place (TOB-YOLO-11); in its current state, codebase regressions are probable, and other implementation flaws are likely to exist.

Second, we identified multiple issues related to the unnecessary use of dangerously permissive functions, such as `subprocess.check_output` (TOB-YOLO-1), `eval` (TOB-YOLO-3), and `os.system` (TOB-YOLO-5). Although the difficulty of these issues is high, an attacker who successfully exploits them could obtain remote code execution, which is an unacceptable risk for security-critical systems.

Recommendations

Based on the lightweight threat modeling exercise, codebase maturity evaluation, and findings identified during the security review, Trail of Bits recommends taking the following steps to better secure the YOLOv7 system:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Implement an adequate testing framework with comprehensive unit tests and integration tests.** Multiple findings in this report could have been prevented with a set of unit tests covering both the happy and sad code paths. In its current state of no unit testing, we believe that other implementation flaws are likely present and code regressions are likely to occur.
- **Remove the use of highly permissive functions, such as `subprocess.check_output`, `eval`, and `os.system`.** As discussed in the individual findings, many instances of these risky functions are unnecessary and have a safer alternative. Each of these instances should be carefully reviewed and strongly considered for removal, as their presence introduces a level of risk unacceptable for security-critical systems.
- **Make improvements to the development process of the codebase**. Code scanning tools such as Semgrep, CodeQL, and bandit uncovered multiple security issues and code quality issues that are included in this report. Including tools such as these in the development process will help prevent similar issues from occurring in the future. In addition, a proper testing framework and contribution guidelines for the codebase will help prevent codebase regressions.
- **Enforce the usage of secure protocols when available.** We note instances where HTTP is used instead of HTTPS. In addition, secure versions of the RMTP/RTSP protocols (RTMPS and RTSPS) should be supported.
- **Keep dependencies as updated as possible to ensure upstream security fixes are applied**. As identified during the lightweight threat model, malicious code or vulnerabilities in various dependencies used by YOLOv7 are an important threat scenario to consider. Tools such as pip-audit can be integrated into the development process to make this process automatic.
- **Provide documentation to users about the potential threats when using data from untrusted training data or webcam streams**. As discussed, we believe that YOLOv7 is currently not suited for use in security-critical systems. Users should be warned that the system does not sufficiently protect against inputs being obtained from external sources (such as training data and webcam streams).

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	5
Medium	2
Low	4
Informational	1

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	1
Data Validation	7
Denial of Service	3
Testing	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Brooke Langhorne, Project Manager
brooke.langhorne@trailofbits.com

The following engineers were associated with this project:

Alvin Crighton, Consultant
alvin.crighton@trailofbits.com

Anusha Ghosh, Consultant
anusha.ghosh@trailofbits.com

Heidy Khlaaf, Consultant
heidy.khlaaf@trailofbits.com

Jim Miller, Consultant
james.miller@trailofbits.com

Kurt Willis, Consultant
kurt.willis@trailofbits.com

Maciej Domanski, Consultant
maciej.domanski@trailofbits.com

Spencer Michaels, Consultant
spencer.michaels@trailofbits.com

Suha Hussain, Consultant
suha.hussain@trailofbits.com

William Woodruff, Consultant
william.woodruff@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
October 31, 2023	Publication of comprehensive report

Project Goals

The engagement was scoped to provide a security assessment of the YOLOv7 codebase. Specifically, we sought to answer the following non-exhaustive list of questions:

- Does the codebase have any security issues that would enable any of the threats listed in the threat scenarios section of Trail of Bits's threat modeling exercise?
- Can a malicious dataset or model file be used to exploit the system?
- Can a network insider or local attacker use their access to compromise a system running YOLO?
- Can an attacker execute code remotely on a target's machine through a compromised dataset, model, or configuration files?
- Does the codebase implement complex arithmetic correctly and safely? Is it properly tested?
- Are the YOLO codebase dependencies up to date?
- Does the codebase adhere to the best practices for Python codebases?
- Can any improvements be made to the CI/CD of the project?

Project Targets

The engagement involved a review and testing of the following target.

YOLOv7

Repository	https://github.com/WongKinYiu/yolov7
Version	3b41c2c
Type	Real-time object detector
Platform	Python

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. We performed a lightweight threat model of the entire system. We then used the results from our lightweight threat model to inform areas of the codebase to review during our secure code review. Our approaches for code review included the following:

- **Triton inference server**. We used Semgrep and CodeQL to automatically identify code quality issues as well as potential security issues. We also performed a manual review of this component, focusing on how the deployment processes, validates, and sanitizes user or external data inputs.
- **Models**. We used Semgrep and CodeQL to automatically identify code quality issues as well as potential security issues. We performed a high-level manual review of this component, focusing on best development practices for Python.
- **Utilities**. We used Semgrep and CodeQL to automatically identify code quality issues as well as potential security issues. We performed a manual review of various critical utility functions that are used throughout the codebase, such as the functions for parsing configuration files and for downloading external datasets, since these were threat scenarios identified during the threat modeling exercise.
- **Main model scripts (`detect.py`, `export.py`, `hubconf.py`, `test.py`, `train_aux.py`, `train.py`)**. We used Semgrep and CodeQL to automatically identify code quality issues as well as potential security issues. We also performed a manual review of this component, focusing on how the user or external data is processed, validated, and sanitized before use. We also focused on verifying that this component follows best development practices for Python.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Models implementation correctness**. We performed a best-effort review of the models component that focused on verifying best development practices for Python. There was not sufficient time to perform a detailed review of every model component that the YOLO system has implemented. Ideally, with more time, these implementations can be reviewed against their original specifications in academic papers or other sources of documentation. Due to the amount of these components in the system, the complexity of these components, and the lack of comprehensive

inline documentation, performing such a review in this engagement was not possible.

- **Utilities**. Our review of the utilities component was a best-effort review that targeted the most critical components identified during the threat modeling exercise. The other areas of the utilities component, such as those handling logging and plotting, would benefit from a more detailed code review.

Lightweight Threat Model

As part of the audit, Trail of Bits conducted a lightweight threat model of YOLOv7, drawing from Mozilla's "Rapid Risk Assessment" methodology and the National Institute of Standards and Technology's (NIST) guidance on data-centric threat modeling (NIST 800-154). We began our assessment of the design of YOLOv7 by reviewing the various YOLO academic papers and user documentation.

Data Types

- Inputs
  - YAML files (--data data/coco.yaml)
  - PT files - initial weights (--weights)
  - TXT files
  - Bash files
  - JSON
  - ZIP
  - Images ('bmp', 'jpg', 'jpeg', 'png', 'tif', 'tiff', 'dng', 'webp', 'mpo')
  - Videos ('mov', 'avi', 'mp4', 'mpg', 'mpeg', 'm4v', 'wmv', 'mkv')
  - HTTP
  - HTTPS
    - Certificate files
  - rtsp/rtmp
- Outputs
  - TorchScript
  - CoreML
  - TorchScript-Lite
  - ONNX

Data Flow

The following diagram shows how data flows through the YOLO system, in the case in which a YOLO model is trained on an external dataset and then deployed to an inference server, where it receives novel inputs from the public Internet. Dotted areas indicate threat zones; red bubbles indicate the positions of threat actors within the system.

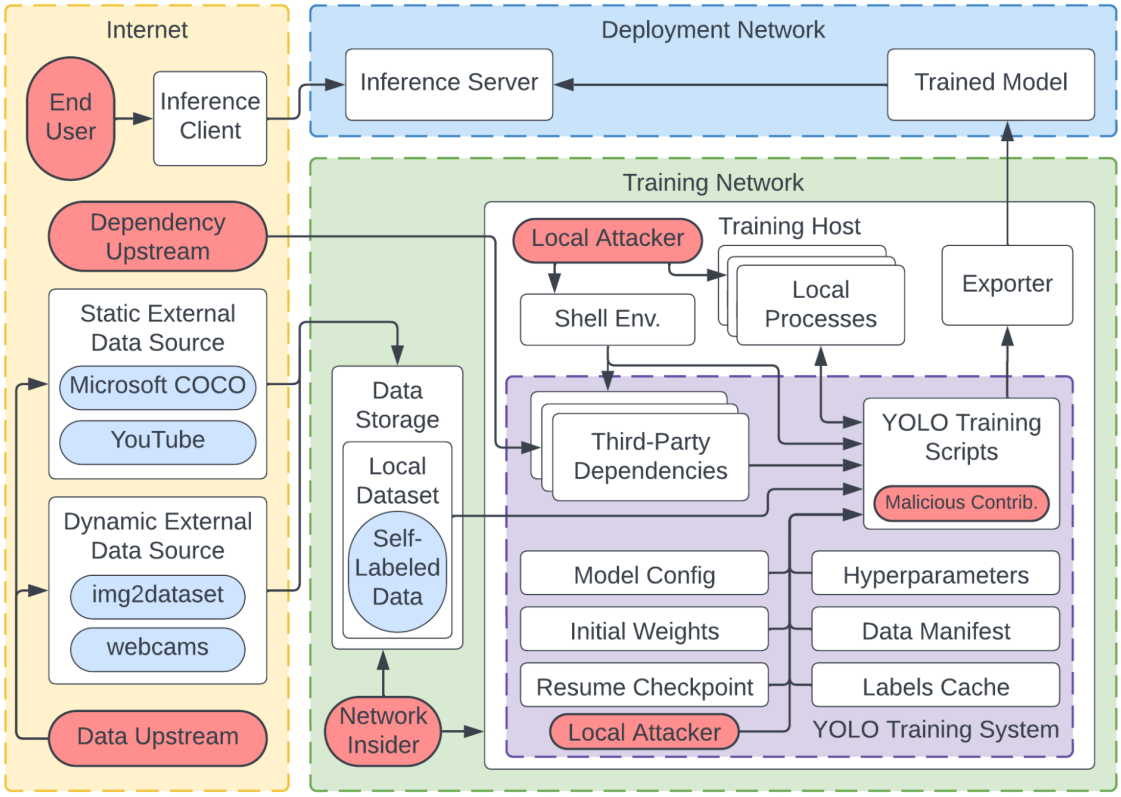

Figure 1: The data flow in a typical YOLO training and deployment scenario

Components and Trust Zones

The following table describes each YOLO component identified for our analysis. It also indicates whether the component or dependency is out of scope for the assessment. We explored the implications of threats involving out-of-scope components that directly affect in-scope components, but we did not consider threats to out-of-scope components themselves.

Component	Description
Internet	The public Internet, accessible to anyone.
Static External Data Source	A source of training, testing, or live data hosted on the public internet that is assembled once and then re-hosted without significant alteration (e.g., Microsoft COCO or specific YouTube videos).
Dynamic External Data Source	A source of training, testing, or live data hosted on the public internet that is assembled dynamically from numerous third-party sources or collected via live image/video capture (e.g., img2dataset and webcams accessed over the Internet.)
Inference Client	A client that communicates with the Inference Server to supply inputs for classification.
Deployment Network	The network in which the trained model is deployed.
Trained Model	The primary output artifact of the YOLO Training System, a model with its weights adjusted according to training data.
Inference Server	The platform (e.g., Triton) on which the Trained Model runs, receiving novel inputs to classify.
Training Network	The network within which a model is trained.
Training Host	The host on which the model is trained.
Data Storage	The host on which the Dataset is stored.
Local Dataset	A source of image or video data that is colocated with the model host (e.g., a physical storage medium or a NAS on the model

	network).
Shell Environment	The shell environment in which the Yolo Scripts are launched.
Exporter	A suite of scripts packaged with YOLO, used to transform raw trained model output into a format usable by the inference server.
Local Processes*	Other processes on the Model Host that communicate with the YOLO system.
YOLO Training System	The collection of scripts and configuration files used to train and test a YOLO model. An instance of the YOLO system is present on the Training Host, and thus resides within the Training Network.
YOLO Scripts	The suite of scripts used to train and deploy YOLO models.
Model Config File (model.yaml)	The model.yaml configuration files, supplied by the `–cfg` flag.
Initial Weights (.pt) File	Initial weights file, .pt, supplied via `–weights`.
Resume Checkpoint File	State file used to resume paused training, supplied via `–resume`.
Hyperparameters File	Hyp.yaml, part of run settings, supplied via `–hyp`.
Data Manifest File	Manifest for training data, supplied via `–data`.
Labels Cache File	Cached image labels.
Third-Party Dependencies*	A locally imported dependency retrieved from a third-party source such as PyPi or a GitHub repository.

Trust Zone Connections

We can draw from our understanding of what data flows between trust zones and why to enumerate attack scenarios.

Origin Zone	Dest. Zone	Data Description	Connection Type	Auth Type
Internet	Internet	Images are scraped from disparate sources into a centralized dataset.	HTTP(S)	None
Internet	Model Network	Datasets are downloaded to a location accessible by the model host in preparation for training.		
Internet	Model Network, YOLO System	Third-party dependencies of the YOLO system are retrieved and installed onto the model host.	HTTPS	None
Internet	YOLO System	Configuration files are downloaded from the internet and loaded by the YOLO scripts.		
Model Network	YOLO System	Configuration files located on the model host's local filesystem, or other hosts on the model network, are loaded by the YOLO scripts.	Local filesystem	Filesystem permissions
Model Network	Model Network	A third party process such as TensorBoard or WandB, running on the model host or an adjacent host, retrieves information generated by YOLO scripts on the model host.	Local filesystem; local sockets	Filesystem permissions
Model Network	YOLO System	Environment variables on the model host are loaded into YOLO's PyTorch execution environment.	POSIX APIs	Local user scope
YOLO System	Model Network	A model is generated via training and deployed to a host.		

Threat Actors

The following table describes actors who could be malicious, could be induced to undertake an attack, or could be impacted by an attack. Defining these actors is helpful in determining which protections, if any, are necessary to mitigate or remediate a vulnerability.

Actor	Description
End User	A user who can submit live data to a deployed model.
Dependency Upstream	The source(s) from which software dependencies are retrieved.
Data Upstream	The source(s) from which image/video data are retrieved.
Network Insider	An attacker with access to at least one host within the model network (not necessarily the model host).
Local Attacker	An attacker with local access to the model host.
Malicious Contributor	An attacker with full or partial control over YOLO's source code.

Threat Scenarios

The following table describes possible threat scenarios given the design, architecture, and risk profile of the YOLO system.

Threat	Scenario	Actor(s)	Component(s)
Dataset Compromise	An attacker gains control of the server hosting a dataset used by YOLO (or performs a Man-in-the-Middle attack between it and the model network) and serves a modified version of it.	• Data Upstream	• Static External Data Source
	An attacker compromises local storage for an already-downloaded model or self-labeled data.	• Network Insider	• Local Dataset
	A domain scraped by img2dataset expires and falls under an attacker's control, allowing the attacker to poison a small portion of the dataset.	• Data Upstream	• Dynamic External Data Source
	An attacker introduces malicious code into a dependency used in the pre-training image processing pipeline, mutating the dataset.	• Dependency Upstream or Local Attacker	• Local Dataset
	An attacker conducts a Man-in-the-Middle attack against an insecure HTTP connection used to download training data, mutating the data in transit.	• Data Upstream	• Static & Dynamic External Data Sources
	An attacker poisons the dataset upstream, leading to mislabeled data in the local dataset.	• Data Upstream	• Static & Dynamic External Data

			Sources
Host Compromise	An attacker gains control of a low-privileged process on the model host, either by connecting to it from another host on the model network, or by compromising its upstream externally.	• Network Insider; External Attacker	• Model Host; Local Process
	An attacker with a foothold on the training host injects shell environment variables which are loaded into PyTorch's init process.	• Local Attacker	• Model Host; Local Process
	An attacker sneaks malicious code into the codebase of the YOLO system or one of its dependencies, gaining control of the host machine.	• Malicious Contributor; Dependency Upstream	• YOLO Scripts; Third Party Dependencies
YOLO Process Compromise	An attacker with local access to the model host exploits a vulnerability in the YOLO system (e.g., by injecting executable code into a configuration file).	• Local Attacker	• Model Config File; Shell Environment
	An attacker compromises a local process such as WandB or Tensorboard and writes to the YOLO training scripts' intermediate state files, corrupting the model weights.	• Local Attacker	• Resume Checkpoint; Labels Cache
	An attacker sneaks malicious code into the codebase of the YOLO system or one of its dependencies, introducing behavior that adversely affects the model's predictions.	• Malicious Contributor; Dependency Upstream	• YOLO Scripts; Third Party Dependencies

	An attacker supplies a malformed image or video file that exploits a vulnerability when processed by the YOLO Scripts or one of their dependencies.	• Data Upstream	• YOLO Scripts; Third Party Dependencies

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use open-source static analysis, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix D
CodeQL	A code analysis engine developed by GitHub to automate security checks	Appendix D
TorchScript Automatic Trace Checker	A dynamic analysis tool included in PyTorch that automatically finds potential errors in traced models	Appendix D

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The codebase includes complicated arithmetic for various models it provides. Unfortunately, the files that implement this arithmetic are poorly documented, untested, and the arithmetic mixes types between integer, floating point, and various tensor types, which is error prone. In addition, we identified an issue (TOB-YOLO-9) related to potential division-by-zero errors that would halt the execution of the triton inference server.	**Weak**
Auditing	Many locations use logging with helpful descriptions to track errors. However, more areas of the codebase should be designed defensively, with better input validation and sanitization and corresponding auditing for when issues are identified.	**Satisfactory**
Authentication / Access Controls	This category is largely unconsidered, since many of the components of YOLOv7 do not require access controls. However, YOLOv7 relies on data and model files that are potentially downloaded from outside sources, and little authentication or verification is performed on these external data sources.	**Moderate**
Complexity Management	Many functions in the codebase are separated with a clear purpose. However, the most complex portion of the codebase, which implements different components of the model architecture (such as neural net layers), is poorly documented and contains large, complex functions that could be divided into multiple functions.	**Moderate**
Configuration	This category is largely unconsidered, but the codebase is not configured to enforce HTTPS connections everywhere	**Moderate**

	(TOB-YOLO-6), and malicious configuration files could result in arbitrary code execution (TOB-YOLO-3).	
Cryptography and Key Management	This category is largely unconsidered, but HTTPS connections are not enforced everywhere (TOB-YOLO-6).	**Moderate**
Data Handling	The codebase performs very little input validation and sanitization, even for data that could be pulled from external sources. This lack of input validation could result in denial of service (TOB-YOLO-9). In addition, the codebase relies on insecure pickle files that could also be obtained from external sources (TOB-YOLO-2).	**Weak**
Documentation	In addition to multiple academic papers describing the YOLO models and their architectures, YOLO also has a comprehensive set of user documentation. Furthermore, the codebase contains docstrings and inline documentation in many locations; however, multiple functions are missing docstrings, and some locations of the codebase have limited inline comments. In particular, the models component of the codebase would benefit the most from additional inline documentation.	**Satisfactory**
Maintenance	The codebase does not have any mechanisms to protect itself from regressions during the development process, such as code scanning, testing, or even contribution guidelines. Since the system does not use any unit tests, we expect bugs to exist in the codebase as well.	**Weak**
Memory Safety and Error Handling	Since the codebase is mostly Python, memory safety is largely unconsidered. Many locations in the codebase catch errors or raise exceptions and report the issues with clear logging. However, some areas could improve error handling, especially components accepting data from potentially external sources (TOB-YOLO-9).	**Moderate**
Testing and Verification	Aside from testing the full model with a test dataset, the codebase is missing a test suite altogether. The codebase would greatly benefit from a comprehensive set of unit tests covering both the happy and sad paths, which would have prevented multiple issues in the codebase. In addition, the codebase would benefit from integration	**Missing**

	testing of the interactions of different models and components, especially for the parts of the codebase that accept outside input, such as model files and data sources.	

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Multiple uses of subprocess.check_output with shell=True could allow command injection	Data Validation	High
2	Models are stored and loaded as pickle files throughout the YOLO codebase	Data Validation	High
3	Parsing of YAML config file can lead to arbitrary code execution	Data Validation	High
4	Untrusted pre-trained models can lead to arbitrary code execution	Data Validation	High
5	Multiple uses of os.system could allow command injection	Data Validation	High
6	Use of unencrypted HTTP protocol	Cryptography	Low
7	Insecure origin check	Data Validation	Low
8	The check_dataset function downloads and unzips files from arbitrary URLs	Denial of Service	Low
9	Insufficient input validation in triton inference server could result in uncaught exception at runtime	Denial of Service	Medium
10	Improper use of TorchScript tracing leads to model differentials	Data Validation	Medium
11	Project lacks adequate testing framework	Testing	Informational

12	Flaw in detect.py will cause runtime exceptions to occur when using a traced model	Denial of Service	**Low**

Detailed Findings

1. Multiple uses of subprocess.check_output with shell=True could allow command injection

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YOLO-1
Target: `utils/torch_utils.py`, `utils/general.py`, `utils/google_utils`	

Description

Various parts of the codebase rely on various shell commands to obtain relevant information for the user. For instance, as shown in figure 1.1, the `git_describe` function uses `subprocess.check_output` to run a git command. Functions such as `subprocess.check_output` are permissive functions that allow arbitrary commands to be run; as a result, it is important that these functions are used carefully to prevent command injection attacks, where an attacker crafts malicious input that results in `subprocess.check_output` running a malicious command.

```
54    def git_describe(path=Path(__file__).parent):  # path must be a directory
55        # return human-readable git description, i.e. v5.0-5-g3e25f1e
https://git-scm.com/docs/git-describe
56        s = f'git -C {path} describe --tags --long --always'
57        try:
58            return subprocess.check_output(s, shell=True,
stderr=subprocess.STDOUT).decode()[:-1]
59        except subprocess.CalledProcessError as e:
60            return ''  # not a git repository
```

Figure 1.1: Snippet of `git_describe` in `utils/torch_utils.py`

```
72    def check_git_status():
73        # Recommend 'git pull' if code is out of date
74        print(colorstr('github: '), end='')
75        try:
76            assert Path('.git').exists(), 'skipping check (not a git repository)'
77            assert not isdocker(), 'skipping check (Docker image)'
78            assert check_online(), 'skipping check (offline)'
79
80            cmd = 'git fetch && git config --get remote.origin.url'
81            url = subprocess.check_output(cmd,
shell=True).decode().strip().rstrip('.git')  # github repo url
82            branch = subprocess.check_output('git rev-parse --abbrev-ref HEAD',
```

```
shell=True).decode().strip()  # checked out
83            n = int(subprocess.check_output(f'git rev-list {branch}..origin/master
--count', shell=True))  # commits behind
84            if n > 0:
85                s = f"⚠️ WARNING: code is out of date by {n} commit{'s' * (n >
1)}. " \
86                    f"Use 'git pull' to update or 'git clone {url}' to download
latest."
87            else:
88                s = f'up to date with {url} ✅'
89            print(emojis(s))  # emoji-safe
90        except Exception as e:
91            print(e)
```

Figure 1.2: Snippet of `check_git_status` in `utils/general.py`

It is recommended that functions like `subprocess.check_output` and `subprocess.run` are called with the input command parameterized in an array (rather than as a single string) and with `shell=False` (the default). The reason for this is that when `shell=False`, these subprocess functions will execute only if each element in the parameterized input array does not contain whitespace. This will prevent any sort of command injection attack, even when the attack can control some of the values in the parameterized input. However, as shown in figures 1.1 and 1.2, multiple locations in the YOLOv7 codebase call `subprocess.check_output` with a single string for the command and `shell=True`.

Here are all the instances of `subprocess.check_output` being called with `shell=True`:

- `utils/general.py` lines 81, 82, 83, and 114
- `utils/google_utils.py` lines 15 and 31
- `utils/torch_utils.py` line 58

Exploit Scenario

An attacker crafts a malicious command that they would like to inject into an instance of `subprocess.check_output`. This attacker forces their target victim to use a directory path or a git branch that contains this malicious command as a substring, which allows them to inject a command into `subprocess.check_output` in either figure 1.1 or figure 1.2.

Recommendations

Short term, call `subprocess.check_output` with `shell=False` in all instances. Also, use a parameterized input array rather than constructing a single string for the command being called.

Long term, review all instances of `subprocess`, `eval`, `os.system`, and any other permissive functions to ensure they are being used safely. In addition, consider replacing these instances with safer internal Python API calls. For instance, consider using GitPython rather than using `subprocess.check_output` to obtain git information.

2. Models are stored and loaded as pickle files throughout the YOLO codebase

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YOLO-2
Target: `detect.py`, `hubconf.py`, `train.py`, `train_aux.py`, `utils/aws/resume.py`, `utils/datasets.py`, `utils/general.py`	

Description

Throughout the YOLOv7 codebase, models are serialized and loaded using functions such as `torch.load` and `torch.save`, which rely on pickle files. Pickle files have become prevalent in the machine learning space for serializing models because their flexibility makes it possible to serialize several kinds of models without much effort. However, pickle files are known to be insecure, as they allow the execution of arbitrary code. If any of these pickle files are obtained from an untrusted source, an attacker could inject malicious code into the pickle file, which would run on the victim's machine.

Figure 1.1 shows one of several locations that rely on `torch.load` to load models. This instance is particularly risky because the model can potentially be downloaded from an external source using the `attempt_download` function. If an attacker is able to compromise the site that hosts these models, they would obtain a vector for remote code execution.

```
def create(name, pretrained, channels, classes, autoshape):
    """Creates a specified model

    Arguments:
        name (str): name of model, i.e. 'yolov7'
        pretrained (bool): load pretrained weights into the model
        channels (int): number of input channels
        classes (int): number of model classes

    Returns:
        pytorch model
    """
    try:
        cfg = list((Path(__file__).parent / 'cfg').rglob(f'{name}.yaml'))[0]  #
model.yaml path
        model = Model(cfg, channels, classes)
        if pretrained:
            fname = f'{name}.pt'  # checkpoint filename
            attempt_download(fname)  # download if not found locally
```

```
ckpt = torch.load(fname, map_location=torch.device('cpu'))  # load
```

Figure 2.1: snippet of create in hubconf.py

We consider this issue to have high difficulty because, in order to exploit it, an attacker must be able to serve a malicious pickle file to a target victim. It is possible that if an attacker is able to serve these malicious files, then they likely have the ability to perform other attacks directly, although this may not always be possible. Moreover, malicious pickle files are much more difficult to detect without proper inspection, as it is possible for these files to execute malicious code and still correctly load the model files.

Exploit Scenario

An attacker serves a malicious pickle file that exfiltrates all of the victim's credentials and sends them to a server controlled by the attacker. The attacker carefully crafts the pickle file so that after the credentials have been exfiltrated, the YOLO model still loads correctly, and the victim does not detect anything malicious.

Recommendations

Short term, when loading PyTorch models, use the `weights_only` unpickler and `load_state_dict()`; consider using fickling to detect possible malicious pickle files before loading them.

Long term, use a safer serialization format, such as safetensors or ONNX, which allows for the serialization of complex models without allowing for the execution of arbitrary code.

References

- Never a dill moment: Exploiting machine learning pickle files

3. Parsing of YAML config file can lead to arbitrary code execution

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YOLO-3
Target: `train.py`	

Description

When initiating a `Model` class in `train.py`, it can take a YAML as a configuration file for the backbone of a model architecture. The configuration file is parsed by the `parse_model` function in `models/yolo.py`. The function uses an `eval` function on lines in the configuration file, as shown in figure 3.1.

```
742    layers, save, c2 = [], [], ch[-1]
743    for i, (f, n, m, args) in enumerate(d['backbone'] + d['head']):
744        m = eval(m) if isinstance(m, str) else m
745        for j, a in enumerate(args):
746            try:
```

Figure 3.1: Snippet of `parse_model` in `models/yolo.py`

The `eval` function allows execution of arbitrary expressions from a string. Without proper validation of inputs, an attacker could inject malicious code to execute on a victim's machine. In `model.py`, the function checks only if the current instance is a string and inputs it to `eval` without any proper validation.

Exploit Scenario

An adversary replaces a list of numbers with a list of a single, malicious string in the backbone section of the configuration file while keeping the rest of the configuration file the same, as shown in figure 3.2.

```
13    backbone:
14        [[-1, 1, Conv, ["__import__('os').system('/bin/sh')"]],
15         [-1, 1, Conv, [64, 3, 2]],
16         [-1, 1, Bottleneck, [64]],
17         [-1, 1, Bottleneck, [64]],
```

Figure 3.2: Snippet of a malicious configuration YAML file

When given this configuration file, the `parse_model` function evaluates the string as code and executes it. In this example, `os.system` was used to open a shell. When the user trains their data in `train.py`, they load this YAML file using the `cfg` flag in the command line.

```
Unset
python3 train.py --workers 8 --device 0 --batch-size 32 --data data/coco.yaml
--img 640 640 --cfg cfg/baseline/yolov7-malicious.yaml --weights '' --name
yolov7 --hyp data/hyp.scratch.p5.yaml
```

Recommendations

Short term, remove the usage of `eval` entirely and instead either construct the objects explicitly or use a modeling library such as Pydantic.

Long term, review all instances of `subprocess`, `eval`, `os.system`, and any other permissive functions to ensure they are being used safely. In addition, consider replacing these instances with safer internal Python API calls, such as those described in the short term recommendation.

4. Untrusted pre-trained models can lead to arbitrary code execution	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YOLO-4
Target: `train.py`	

Description

The same vulnerability mentioned in finding TOB-YOLO-3 can be exploited using a pretrained model. In `train.py`, the user is allowed to provide a configuration YAML file as architecture backbone or a `*.pt` file as a pretrained model.

```
88      model = Model(opt.cfg or ckpt['model'].yaml, ch=3, nc=nc,  \
            anchors=hyp.get('anchors')).to(device)
```

Figure 4.1: Creating a model class using the pretrained file

The pretrained model can have an attribute called `yaml`, which is similar to the YAML used for the model architecture except that it is a Python dictionary. Since the parsing is the same, the vulnerability is still present, and the `eval` function can be exploited by crafting a malicious pretrained file.

Exploit Scenario

The attacker writes their own `Model` class with an attribute called `yaml` that is a dictionary with the same properties as a YAML file used as a configuration file. The victim user creates an instance of that class and stores it in a dictionary with the string `'model'` as a key and the object as its value. The victim user then saves the dictionary as a `*.pt` file using PyTorch, which creates a malicious pretrained file. The victim user then loads the pretrained file in the command line using the `weights` flag.

```
Unset
python3 train.py --workers 8 --device 0 --batch-size 32 --data
data/coco.yaml --img 640 640 --weights 'maliciousyolov7.pt'
--name yolov7 --hyp data/hyp.scratch.p5.yaml
```

This exploit is difficult to detect due to the serialization of the object.

Recommendations

Short term, support only pretrained weights from the GitHub repo with a checksum to ensure the downloaded pretrained file is not malicious.

Long term, use a safer serialization format, such as safetensors or ONNX, which allows for the serialization of complex models without allowing for the execution of arbitrary code.

5. Multiple uses of `os.system` could allow command injection

Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YOLO-5
Target: `train.py`, `train_aux.py`, `test.py`, `utils/general.py`, `utils/google_utils.py`	

Description

The codebase uses Python's `os.system` to invoke certain commands. These are susceptible to malicious command injections.

Certain commands, such as `gsutil`, `unzip` and `curl`, are executed as shell commands via Python's `os.system`. An example can be found in the `train.py` script.

```
if opt.bucket:
    os.system(f'gsutil cp {final} gs://{opt.bucket}/weights')  # upload
```

Figure 5.1: Use of `os.system` in the `train` function in `train.py`

This particular use of `os.system` is vulnerable to command injection. By including the command line argument `--bucket ";whoami"`, we can invoke arbitrary commands as the current user.

The full command to be executed might look like this:

```
python train.py  --bucket ";whoami" --data data/coco.yaml --img 640 640 --cfg
cfg/training/yolov7.yaml --weights '' --name yolov7 --hyp data/hyp.scratch.p5.yaml
--epochs 1
```

Figure 5.2: Example command line injection for `train.py`

Another problematic instance of `os.system` is in the `check_dataset` function:

```
def check_dataset(dict):
    # Download dataset if not found locally
    val, s = dict.get('val'), dict.get('download')
    if val and len(val):
        val = [Path(x).resolve() for x in (val if isinstance(val, list) else [val])]
# val path
        if not all(x.exists() for x in val):
            print('\nWARNING: Dataset not found, nonexistent paths: %s' % [str(x)
for x in val if not x.exists()])
            if s and len(s):  # download script
```

```
                print('Downloading %s ...' % s)
                if s.startswith('http') and s.endswith('.zip'):  # URL
                    f = Path(s).name  # filename
                    torch.hub.download_url_to_file(s, f)
                    r = os.system('unzip -q %s -d ../ && rm %s' % (f, f))  # unzip
                else:  # bash script
                    r = os.system(s)
                print('Dataset autodownload %s\n' % ('success' if r == 0 else
'failure'))  # analyze return value
            else:
                raise Exception('Dataset not found.')
```

Figure 5.3: Use of `os.system` in the `check_dataset` function in `utils/general.py`

As highlighted in figure 5.3, the `check_dataset` function takes in a dictionary, which contains a download script `s`. If `s` is not a URL containing the substring `` `http` `` and `` `zip` ``, then `check_dataset` will execute an arbitrary command by calling `os.system(s)`. This is particularly problematic because this function is called in both `train.py` and `test.py` with the dictionary value obtained from a YAML file specified by the user. If an attacker was able to compromise such a YAML file, this would introduce an arbitrary code execution vulnerability.

Throughout the repository, there are many uses of `os.system`, many of which (but not all) are susceptible in the same way:

- test.py#L352
- train_aux.py#L430
- train_aux.py#L512
- train_aux.py#L656
- train.py#L433
- train.py#L515
- train.py#L662
- utils/general.py#L168
- utils/general.py#L170
- utils/general.py#L826
- utils/general.py#L844
- utils/google_utils.py#L47
- utils/google_utils.py#L67
- utils/google_utils.py#L72
- utils/google_utils.py#L84
- utils/plots.py#L410
- utils/aws/resume.py#L37

Exploit Scenario

The YOLOv7 repository is deployed as a cloud service for paying customers. Eve, a malicious user, spots the vulnerability and injects a command that starts a remote shell

execution environment that she can access from her computer. She is now in control of the servers.

Recommendations

Short term, be vigilant when handling user-provided inputs. Heavily limit or sanitize these in order to reduce the expressivity of inputs. Extra care should be given to strings or inputs of arbitrary length, especially when these are being used in combination with commands that are able to execute arbitrary commands.

Long term, review all instances of `subprocess`, `eval`, `os.system`, and any other permissive functions to ensure they are being used safely. In addition, consider replacing these instances with safer internal Python API calls.

6. Use of unencrypted HTTP protocol	
Severity: **Low**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-YOLO-6
Target: `README.md`	

Description

YOLOv7 uses the unencrypted HTTP protocol in the documentation to download MS COCO dataset images (figure 6.1), which could allow an attacker to intercept and modify both the request and response of a victim user in the same network. The attacker could then manipulate the training set.

```
 83    ## Training
 84
 85    Data preparation
 86
 87    ``` shell
 88    bash scripts/get_coco.sh
 89    ```
 90
 91    * Download MS COCO dataset images
([train](http://images.cocodataset.org/zips/train2017.zip),
[val](http://images.cocodataset.org/zips/val2017.zip),
[test](http://images.cocodataset.org/zips/test2017.zip))
```

Figure 6.1: Part of the YOLOv7 documentation that uses HTTP protocol to download dataset images (`yolov7/README.md#83–91`)

Exploit Scenario

Eve gains access to Alice's network and modifies Alice's downloaded training data to obtain specific recognizing behavior of YOLOv7.

Recommendations

Short term, enforce the use of the HTTPS URL scheme in the YOLOv7 documentation.

Long term, review any YOLOv7 code that contains external links and ensure that those links do not use the HTTP protocol.

7. Insecure origin check	
Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YOLO-7
Target: `utils/datasets.py#285`	

Description

YOLOv7 insecurely checks the origin of the URLs (figure 7.1) by checking for `youtube.com/` or `youtu.be/` anywhere in the URL string. This validation can be bypassed by using any domain and the `youtube.com/` or `youtu.be/` strings as a parameter.

```
285    if 'youtube.com/' in str(url) or 'youtu.be/' in str(url):  # if source is
YouTube video
```

Figure 7.1: Insecure origin check implementation (`yolov7/utils/datasets.py#285`)

We have rated this issue as low severity because it affects the model only during detection. This issue would be more severe if this occurred during training, as this could allow an attacker to poison the training data and perform a backdoor attack or generally degrade the model's performance.

Exploit Scenario

Eve creates a malicious website, `evil.com`, and crafts a URL that passes the application's origin check, such as `evil.com/whatever?evilparam=youtube.com/`. Eve then tricks Alice into using the deceptive link. Alice, who is unaware of the malicious link, downloads a tainted video, and her YOLO model performs very poorly. Alice then loses valuable time attempting to debug her model before realizing the issue was with her video.

Recommendations

Short term, ensure that `youtube.com` or `youtu.be` strings are present in the main domain section of the URL. Be aware of deceptive subdomain usage, as allowed strings can be used as subdomains (e.g., `youtube.com.evil.com`).

Long term, incorporate CodeQL into your development process to avoid incomplete URL substring sanitization.

8. The check_dataset function downloads and unzips files from arbitrary URLs

Severity: **Low**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-YOLO-8
Target: `test.py`, `train_aux.py`, `train.py`	

Description

The `check_dataset` function is used throughout the codebase to see if a dataset exists at a particular directory path; if the dataset does not exist, then the `check_dataset` function attempts to download the dataset by either downloading a zip file or running a bash script specified in the input.

```
 156    def check_dataset(dict):
 157        # Download dataset if not found locally
 158        val, s = dict.get('val'), dict.get('download')
 159        if val and len(val):
 160            val = [Path(x).resolve() for x in (val if isinstance(val, list) else
[val])]  # val path
 161            if not all(x.exists() for x in val):
 162                print('\nWARNING: Dataset not found, nonexistent paths: %s' %
[str(x) for x in val if not x.exists()])
 163                if s and len(s):  # download script
 164                    print('Downloading %s ...' % s)
 165                    if s.startswith('http') and s.endswith('.zip'):  # URL
 166                        f = Path(s).name  # filename
 167                        torch.hub.download_url_to_file(s, f)
 168                        r = os.system('unzip -q %s -d ../ && rm %s' % (f, f))  #
unzip
 169                    else:  # bash script
 170                        r = os.system(s)
 171                    print('Dataset autodownload %s\n' % ('success' if r == 0
else 'failure'))  # analyze return value
 172                else:
 173                    raise Exception('Dataset not found.')
```

Figure 8.1: `check_dataset` function downloads and unzips from arbitrary URLs

This could be highly problematic in some instances. For example, in the `test.py` file, this function is called on the data variable that is obtained from reading a YAML file that is specified via a command line argument. If this YAML file is corrupted, an attacker could inject a URL that will result in the target user unzipping a zip bomb that halts execution of the model.

Exploit Scenario

A malicious actor corrupts the dataset YAML file being used by a target user during training and injects a malicious download URL. The target user does not inspect their YAML files closely and unknowingly downloads and unzips a zip bomb that halts execution of their model.

Recommendations

Short term, validate the zip file before unzipping to prevent a zip bomb attack. For example, check the size of the file and do not unzip it if it is too large.

Long term, limit which URLs users can download files from or carefully verify that downloaded files can be trusted before unzipping them.

9. Insufficient input validation in triton inference server could result in uncaught exception at runtime

Severity: **Medium**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-YOLO-9
Target: `deploy/triton-inference-server`	

Description

The triton inference server is an open source software that streamlines AI inference. The triton inference server component of the YOLOv7 codebase includes logic for deploying YOLOv7 to the triton inference server. The `client.py` file implements a command line interface for interacting with YOLO models deployed on triton; for example, using this command line interface, users can pass in images and videos to be evaluated.

Despite this command line interface accepting images and videos from potentially external, untrusted sources, very limited input validation is performed on these inputs. As a result, several crafted inputs could cause execution to halt due to uncaught exceptions or other errors. One such example is shown in figure 9.1, where both the `preprocess` and `postprocess` functions (both of which are called in `client.py`) have multiple locations where a division-by-zero error could occur.

```
 6     def preprocess(img, input_shape, letter_box=True):
 7         if letter_box:
 8             img_h, img_w, _ = img.shape
 9             new_h, new_w = input_shape[0], input_shape[1]
 10            offset_h, offset_w = 0, 0
 11            if (new_w / img_w) <= (new_h / img_h):
 12                new_h = int(img_h * new_w / img_w)
 13                offset_h = (input_shape[0] - new_h) // 2
 14            else:
 15                new_w = int(img_w * new_h / img_h)
 16                offset_w = (input_shape[1] - new_w) // 2
 17            resized = cv2.resize(img, (new_w, new_h))
 18            img = np.full((input_shape[0], input_shape[1], 3), 127,
dtype=np.uint8)
 19            img[offset_h:(offset_h + new_h), offset_w:(offset_w + new_w), :] =
resized
 20        else:
 21            img = cv2.resize(img, (input_shape[1], input_shape[0]))
 22
 23        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
 24        img = img.transpose((2, 0, 1)).astype(np.float32)
 25        img /= 255.0
```

```
 26        return img
 27
 28    def postprocess(num_dets, det_boxes, det_scores, det_classes, img_w, img_h,
input_shape, letter_box=True):
 29        boxes = det_boxes[0, :num_dets[0][0]] / np.array([input_shape[0],
input_shape[1], input_shape[0], input_shape[1]], dtype=np.float32)
 30        scores = det_scores[0, :num_dets[0][0]]
 31        classes = det_classes[0, :num_dets[0][0]].astype(np.int)
 32
 33        old_h, old_w = img_h, img_w
 34        offset_h, offset_w = 0, 0
 35        if letter_box:
 36            if (img_w / input_shape[1]) >= (img_h / input_shape[0]):
 37                old_h = int(input_shape[0] * img_w / input_shape[1])
 38                offset_h = (old_h - img_h) // 2
 39            else:
 40                old_w = int(input_shape[1] * img_h / input_shape[0])
 41                offset_w = (old_w - img_w) // 2
 42
 43        boxes = boxes * np.array([old_w, old_h, old_w, old_h], dtype=np.float32)
 44        if letter_box:
 45            boxes -= np.array([offset_w, offset_h, offset_w, offset_h],
dtype=np.float32)
 46        boxes = boxes.astype(np.int)
 47
 48        detected_objects = []
 49        for box, score, label in zip(boxes, scores, classes):
 50            detected_objects.append(BoundingBox(label, score, box[0], box[2],
box[1], box[3], img_w, img_h))
 51        return detected_objects
```

Figure 9.1: `preprocess` and `postprocess` functions do not validate inputs and could trigger division-by-zero errors.

A comprehensive suite of unit tests, covering both the happy and sad codepaths, could help to identify and resolve issues like this. Without any existing unit tests in the triton inference code, it is likely that other crafted input values could result in halting the execution of the inference, or potentially even more severe results.

Exploit Scenario

A malicious actor targets a system using YOLOv7 deployed on triton in which high availability is essential, such as an autonomous vehicle system. An attacker discovers these implementation flaws and causes execution to halt by passing in malformed images, which will result in a division-by-zero error occurring during either pre-processing or post-processing. The system experiences a denial of service.

Recommendations

Short term, update `preprocess`, `postprocess`, and `client.py` to properly handle inputs that currently cause a division-by-zero error to occur.

Long term, implement a comprehensive suite of unit tests to cover both the happy and sad paths of critical components. In addition, consider incorporating static analysis tools like Semgrep and CodeQL into your development process.

10. Improper use of TorchScript tracing leads to model differentials

Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-YOLO-10
Target: `utils/torch_utils.py, detect.py, test.py, export.py`	

Description

To facilitate deployment, Pytorch offers `torch.jit.trace` to convert models into the TorchScript format. However, as shown in table 10.1, there are many known cases in which tracing does not lead to an accurate representation.

Tracer Edge Cases	**Example**
Input-dependent control flow (including mutable container types and in-place operations)	Lines `34`, `35`, `37`, `40`, and `51` of `models/yolo.py`
Certain tensor operations from external libraries and implicit type conversions with tensors	Lines `50-59` of `models/experimental.py`

Table 10.1: Edge cases in which tracing does not produce accurate representation

These cases are present in the defined YOLOv7 models that are currently being traced . This means that the deployed model is different from the original model, yielding different results.

In addition, in this codebase, tracing is performed after the model is serialized to a PyTorch file and then deserialized. This practice, in conjunction with the non-standard structure of the model architecture code, results in the loss of information that analyzes the veracity of training, such as tracer warnings indicating the presence of edge cases.

```
362    traced_script_module = torch.jit.trace(self.model, rand_example,
strict=False)
```

Figure 10.1: Improper use of TorchScript tracing (`yolov7/utils/torch_utils.py#362`)

This use of tracing could introduce differentials that enable the creation of backdoors. For instance, an attacker could craft a malicious model that behaves differently when deployed.

Specifically, this attacker could introduce a backdoor of custom logic that executes only on deployed models.

Exploit Scenario
An attacker trains a model that exhibits a specific, potentially malicious behavior when deployed that is not present otherwise. Specifically, the attacker creates a model that has special behavior for specific input images. Since this behavior is present only for specific images and only during deployment, detecting this backdoored behavior is difficult.

Recommendations
Short-term, mix both tracing and scripting of the model to ensure that all tracing edge cases are avoided. It would also be useful to minimize edge cases, especially those indicated by tracer warnings, to reduce the possibility of differentials. The integrity and effectiveness of tracing should also be tested before serialization by using the automatic trace checker.

Long-term, use `torch.compile` instead of tracing and scripting, as it minimizes the presence of differentials.

11. Project lacks adequate testing framework

Severity: **Informational**	Difficulty: **High**
Type: Testing	Finding ID: TOB-YOLO-11
Target: Y0L0v7	

Description
Currently, the YOLOv7 codebase does not contain any form of testing framework. The only testing of the codebase is performed on the model itself via the typical training and testing that is performed on machine learning models. Notably, there are no other units or integration tests in the codebase.

Unit tests help expose errors and help provide additional documentation or understanding of the codebase to readers. Moreover, they exercise code in a more systematic way than any human can. A strong suite of unit tests is essential to protect against codebase regressions. A stronger testing suite could have prevented the occurrence of multiple issues in this report, such as TOB-YOLO–9, and there are likely other issues in the codebase that could be uncovered by a stronger test suite.

At a minimum, unit tests covering both the happy and sad paths should be added for all critical functions, especially those that accept input from potentially external sources. Ideally, this test suite could be extended to include the entire codebase and also include integration tests that test the interaction between multiple components (again, especially if these components interact with external input).

Exploit Scenario
A security-critical system relies on YOLOv7 for real-time object detection. A malicious actor closely monitors the system and the YOLOv7 codebase. Due to a lack of a testing framework that prevents code regressions, an old, critical flaw is reintroduced into the codebase in a recent commit to the YOLOv7 codebase. The malicious actor identifies this flaw and exploits the security-critical system using this vulnerability.

Recommendations
Long term, implement a comprehensive suite of unit tests to cover both the happy and sad paths of critical components. In addition, consider incorporating static analysis tools like Semgrep and CodeQL into your development process.

12. Flaw in detect.py will cause runtime exceptions to occur when using a traced model

Severity: **Low**	Difficulty: Low
Type: Denial of Service	Finding ID: TOB-YOLO-12
Target: `detect.py`	

Description

The `detect.py` file provides command-line arguments for using the trained YOLO models. These command-line arguments include `--img-size` and `--no-trace`. The former argument controls the size of the image being sent to the model, and the latter controls whether or not a traced model is used. The function `check_img_size()` updates the size of the image (if it is invalid) using the variable `imgz`. However, when the model is traced, the original input size is passed to the model instead of the updated size. This results in a runtime error when an invalid image size is passed when the model is traced that is not present otherwise.

```
34   model = attempt_load(weights, map_location=device)  # load FP32 model
35   stride = int(model.stride.max())  # model stride
36   imgsz = check_img_size(imgsz, s=stride)  # check img_size
37   if trace:
38        model = TracedModel(model, device, opt.img_size)
```

Figure 12.1: Potential runtime exception in `detect.py#L38`

Exploit Scenario

A malicious actor targets a system using YOLO in which high availability is essential, such as an autonomous vehicle system. An attacker discovers that this implementation flaw exists in the version of YOLO being run in the system and causes the target system to attempt to trace a model with invalid image size. Due to this implementation flaw, execution halts and the system experiences a denial of service.

Recommendations

Short term, adjust `detect.py` to resolve this implementation flaw and allow tracing to occur with the proper image size.

Long term, implement a comprehensive suite of unit tests to cover both the happy and sad paths of critical components. In addition, consider incorporating static analysis tools like Semgrep and CodeQL into your development process.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Configuration	The configuration of system components in accordance with best practices
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Data Handling	The safe handling of user inputs and data processed by the system
Documentation	The presence of comprehensive and readable codebase documentation
Maintenance	The timely maintenance of system components to mitigate risk
Memory Safety and Error Handling	The presence of memory safety and robust error-handling mechanisms
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following findings and recommendations are not associated with specific vulnerabilities. However, they enhance code readability or performance and may prevent the introduction of vulnerabilities in the future. These were found using a combination of manual and automated code review.

- **Various parts of the codebase use NumPy arrays inside of PyTorch modules**. It is considered best practice to avoid using NumPy inside of PyTorch because it is more efficient to use the PyTorch equivalent operations. In addition, this can cause issues with tracing, exporting, and distributed training. NumPy is used in the `model/common.py` file on lines 538, 902, 904, 917, 1122, and 1331.
- **Several functions and classes are missing docstrings**. Docstrings enhance code readability and help prevent the introduction of logic bugs. Run CodeQL against the codebase, review all locations that are missing docstrings, and add docstrings to all critical functions and classes.
- **Several files have unused imports and use `import *`**. It is considered best practice to only import modules that are being used and to avoid using `import *` for efficiency and to avoid pollution of the namespace. Run CodeQL against the codebase and review all instances of unused imports and `import *`.
- **Several files contain commented-out code**. It is considered best practice to remove commented-out code that is no longer needed, as it clutters the codebase and diminishes readability. Run CodeQL against the codebase and review all instances of commented-out code.
- **Class and function names do not follow Python naming conventions**. It is best practice to follow the naming conventions for a given programming language when defining things such as classes and functions. Following the standard naming convention will help improve the readability of the codebase. Currently, the `DWConv` function in `models/common.py#L147`, the `autoShape` class in `models/common.py#L865`, and the `aLRPLoss` class in `utils/loss.py#L275`, do not follow standard naming conventions; functions should begin with a lowercase letter, and classes should begin with an uppercase letter.
- **Multiple code paths in the codebase are unreachable**. There are multiple locations in the codebase that contain logic that is unreachable because it is currently behind an `if` statement that will always be `false`. These locations are currently unreachable: lines 47 and 97 of `detect.py`, lines 689 and 786 of `utils/general.py`, and line 45 of `utils/google_utils.py`. All instances that are mistakes should be corrected; otherwise, the logic in these unreachable code paths should be removed.

- **Multiple parameters and variables are unused throughout the codebase**. Multiple locations in the codebase that contain parameters and variables that are defined but never used. It is a best practice to remove all unused values to improve the readability of the codebase. Run CodeQL against the codebase and review all instances of unused parameters and variables.

- **Multiple built-in variables are shadowed by local variables**. The Python language contains several built-in variables, such as `str` and `dict`. However, Python allows you to declare variables with the same name, which will make the built-in variables unusable within the scope of that variable. It is considered a best practice not to name variables after these keywords, as this hurts readability and could lead to unexpected errors. The following locations shadow either built-in variables or global variables: line 956 of `models/common.py`, lines 101 and 269 of `test.py`, lines 57 and 156 of `utils/general.py`, and lines 172 and 216 of `utils/torch_utils.py`.

- **Magic numbers are used throughout the codebase**. It is considered a best practice to not rely on magic numbers as it hurts readability and makes the codebase more error-prone to maintain. Each instance of a magic number should be replaced with a variable of some kind. This can be found in the `model/common.py` file on lines 475, 476, 1090, 1237, and 1238.

- **Assert statements are used throughout the codebase**. This is not recommended, as the corresponding code is removed when compiling to optimized bytecode.

- **Various parts of the codebase utilize Python `set()`**. This is not recommended, as the behavior of sets differs between Python versions with regards to iteration order. This was found in L59 of `export.py`, L262 of `models/experimental.py`, L201 of `test.py`, L54 of `utils/datasets.py`, and L137 of `utils/wandb_logging/wandb_utils.py`.

- **Various parts of the codebase include implicit type conversions with Torch tensors**. The YOLO system has a fairly complex model architecture that requires complex tensor operations performed across multiple layers. Python's flexible type system is known to be error-prone; when implementation flaws occur from mixed types, debugging these issues can be difficult and time consuming. For YOLOv7, these types of issues could result in model performance degradation or denial-of-service attacks. It is considered a best practice to minimize the mixing of different numeric types in order to mitigate these types of issues. For instance, the constructor `torch.Tensor()` is used instead of `torch.tensor()` in multiple places; this is not recommended, as `torch.Tensor()` is a legacy constructor and an alias for `torch.FloatTensor()`, resulting in an implicit type conversion. In addition, tensor operations performed by non-Torch functions, such as NumPy, Math, or built-in Python functions, can also result in implicit type conversions.

Review instances of the codebase where critical operations are being performed, and, where possible, unify the numeric types to a single type, such as FP32.

D. Automated Testing

This section describes the setup of the automated analysis tools used during this audit.

CodeQL

We analyzed the codebase with the public Python CodeQL queries, `codeql-python`. These rules resulted in the discovery of multiple items listed in appendix C. To run CodeQL on the codebase, first create the Python database by running `codeql database create codeql.db –language=python`. Then, run the Python queries by running `codeql database analyze codeql.db –{RULESET}`. If there are any plans to incorporate CodeQL into the YOLOv7 codebase, we recommend reviewing CodeQL's licensing policies.

Semgrep

Semgrep can be installed using `pip` by running `python3 -m pip install semgrep`. To run Semgrep on a codebase, run `semgrep --config "<CONFIGURATION>"` in the root directory of the project. Here, `<CONFIGURATION>` can be a single rule, a directory of rules, or the name of a rule set hosted on the Semgrep registry.

Trail of Bits maintains a repository of custom Semgrep rules targeting the misuse of Machine Learning platforms such as PyTorch. We ran these custom Semgrep rules against the YOLOv7 codebase to identify potential ML-specific issues. These rules resulted in the discovery of issue TOB-YOLO-2 and some of the items listed in appendix C.

TorchScript Automatic Trace Checking

When using `torch.jit.trace`, a randomly generated set of input tensors can be passed into the `check_inputs` argument to utilize the TorchScript automatic tracer checker. The checker retraces the computation of the model on these tensors to find any divergences. In conjunction with this, both the `check_tolerance` and `check_trace` arguments should be used, and the `strict` argument should be set to `True`. This facilitates the pinpointing of edge cases.