Fake Architecture Orchestrator

Alessandra Blasioli
alessandra.blasioli@studio.unibo.it

January 6, 2024

Abstract

The strategic use of *deceptive defense* has been proposed by cybersecurity experts as a method to exploit the typically enjoyed informational asymmetry of attackers, turning it into a tool favoring defenders. This strategy involves creating fake services and components that appear similar to valuable targets for aggressors, aiming to divert attackers' attention and resources away from critical assets. This project aims to easily instantiate a complex fake infrastructure from an architectural diagram, providing a tangible opportunity to implement and evaluate the deceptive defense strategy within the realm of cybersecurity.

1 Introduction

The concept of ***deceptive defense*** in cybersecurity marks a shift in traditional defense mechanisms, aiming to strategically outmaneuver attackers. This approach harnesses the inherent information asymmetry present between attackers and defenders to tilt the advantage towards the defending side.

The crux of this strategy lies in the creation of deceptive elements (simulated services and components) that mirror or imitate genuine high-value assets within an organization's network. These fabricated entities are deliberately designed to appear enticing and valuable to potential attackers, effectively acting as lures.

When attackers engage with these false elements, drawn by their apparent value, they inadvertently divert their attention and expend their resources attempting to breach these decoys. This misdirection tactic serves a dual purpose: it not only stalls attackers' progress but also provides defenders with a window of opportunity to detect, analyze, and respond to the intrusion attempts in real-time.

Moreover, the investment of attackers' time and effort in targeting these counterfeit assets significantly depletes their capability to focus on and penetrate genuinely critical systems or data repositories. As a result, the defenders gain a crucial advantage by safeguarding their core assets, reducing the risk of potential breaches and minimizing the overall impact of cyberattacks.

The ongoing evolution of deceptive defense strategies continues to reshape cybersecurity paradigms, prompting the exploration of innovative methods to proactively thwart cyber threats while bolstering resilience against sophisticated attacks. This approach aligns

with the proactive stance necessary in today's dynamic threat landscape, emphasizing the importance of leveraging deception as a powerful tool in the defender's arsenal.

The primary objective of this project is to automate the creation of a complex simulated structure based on a network schema file, whether it be an image or an XML file. This simulated infrastructure will be modeled following the guidelines provided in the architectural blueprint, replicating the complexity and arrangement of network components.

2 Background

2.1 YOLO (You Only Look Once)

YOLO[1], standing for "You Only Look Once", represents a groundbreaking open-source object detection framework harnessing the capabilities of Convolutional Neural Networks (CNNs). Its exceptional proficiency lies in its rapid object identification and localization within images and videos in real-time. What sets YOLO apart is its efficiency in detecting objects within a single pass through the neural network, ensuring swift and accurate results.

2.1.1 YOLOv5: Advancements in Object Detection

YOLOv5[2], building upon the foundational YOLO (You Only Look Once) framework, marks a significant leap in object detection.

The framework's strength lies in its optimized architecture, effectively balancing speed and accuracy. Notably, YOLOv5 exhibits heightened accuracy in identifying and localizing objects across various scales, ensuring precise detection.

Its simplicity and adaptable nature allow for seamless customization, catering to diverse applications and environments. Overall, YOLOv5 emerges as an attractive choice for object detection endeavors, offering enhanced accuracy, speed, and flexibility to meet real-time application demands.

2.2 Terraform

Developed by HashiCorp, Terraform[3] emerges as a versatile open-source infrastructure orchestration tool. Empowering users with code-based configuration files, Terraform allows for the definition and management of diverse infrastructure resources across various cloud providers and on-premises environments. Its robust capabilities facilitate automated provisioning and systematic resource management, streamlining the deployment process.

[1] YOLO, a brief history
[2] yolov5
[3] terraform.io

2.3 Docker

Functioning as a prominent containerization platform, Docker[4] revolutionizes the creation, distribution, and management of software containers. These containers establish a standardized environment for running applications, abstracting them from the underlying operating system. Docker's prowess lies in encapsulating essential software dependencies within containers, ensuring consistent performance across a spectrum of environments.

2.3.1 Docker Compose

Docker Compose[5], utilizing YAML, defines configurations for multi-container applications. This consolidation streamlines the deployment of various interconnected services by encapsulating essential settings, such as services, networks, and volumes. Integrated within this single file, each service declaration represents a distinct Docker container, facilitating straightforward communication between containers through designated networks or exposed ports.

By encapsulating configurations, Docker Compose simplifies the deployment process. With a single command, it manages the creation, startup, and oversight of containers and resources, minimizing complexities in deployment procedures.

3 Input Data

The input data for this project can come in two forms. We can begin by analyzing an XML file containing a network configuration in XML format, or we can start from an image representing a network configuration.

The XML file was obtained from a diagram created using draw.io[6], an online drawing tool that allows users to create various diagrams, including network diagrams. To obtain the XML version of a network diagram, users can create the diagram using draw.io and then export it in XML format.

```
<mxCell id="69ecfefc5c41e42c-3" value="Firewall" style="image;html=1;labelBackgroundColor=#ffffff;image=img/lib/clip_art/networking/Firewall_
  <mxGeometry x="466.5" y="330" width="80" height="80" as="geometry" />
</mxCell>
<mxCell id="69ecfefc5c41e42c-5" value="PC" style="image;html=1;labelBackgroundColor=#ffffff;image=img/lib/clip_art/computers/Monitor_Tower_12
  <mxGeometry x="370" y="640" width="80" height="80" as="geometry" />
</mxCell>
<mxCell id="69ecfefc5c41e42c-9" value="Laptop" style="image;html=1;labelBackgroundColor=#ffffff;image=img/lib/clip_art/computers/Netbook_128x
  <mxGeometry x="590" y="610" width="80" height="80" as="geometry" />
</mxCell>
<mxCell id="157l2WiuNIdGBpRHh-2j-1" value="" style="image;html=1;image=img/lib/clip_art/computers/Database_128x128.png" vertex="1" parent="1"
```

Figure 1: Section of the XML file used for the conducted experiments, highlights the fields from which information about the service type is extracted.

If the input happens to be an image, it was consistently generated using draw.io, simply by exporting the diagram in JPG format.

[4] docker.com
[5] docs.docker.com/compose/
[6] draw.io

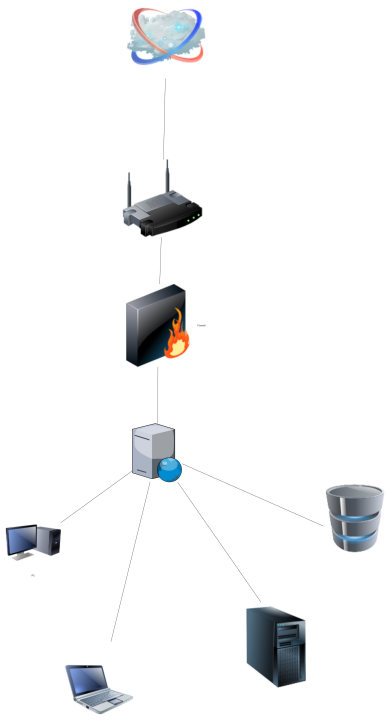

Figure 2: Network diagram schema used as the input image for the experiments.

4 Dataset

In scenarios where the input data manifests as an image, a requisite step involves implementing an element detection process within the image using the YOLO network, as expounded upon in the implementation section. This segment pivots to explicate the dataset preparation crucial for training the YOLO network.

The dataset's origination stemmed from Roboflow[7], a specialized platform adept at facilitating the creation and management of datasets.

Images were methodically sourced from the web to construct an object detection dataset, an imperative aspect aimed at numerically assessing the count of components embedded within the input image. Meticulous individual image labeling was executed, culminating in the classification of data into six distinct classes: Database, Server, Firewall, Web-Server, Client, and Router.

The dataset comprises three main subsets. The *train set* represents the largest portion, constituting 88% of the dataset. It is primarily used for training the model to understand the various components within the images. Following this, the *validation set*, taking up 8% of the dataset, serves to validate the model's performance during the training phase. Lastly, the *test set*, accounting for 4% of the dataset, remains segregated for the final evaluation of the model's overall efficacy and accuracy.

Throughout the preprocessing stage, a series of consistent transformations were systematically applied to standardize the dataset. This involved leveraging Auto-Orient to ensure a consistent alignment of images across all samples. Additionally, a resizing process was employed, resulting in a uniform dimension of 512x512 pixels for all images,

[7] roboflow.com

guaranteeing homogeneity in model training.

Augmentation techniques were deliberately introduced to enrich the dataset's diversity and fortify the model's capabilities. Each training instance underwent three augmentations to introduce variability. These included 90° rotations in both clockwise and counter-clockwise directions, as well as horizontal and vertical shearing of up to ±15°.

These preprocessing measures and augmentations were methodically implemented to enhance the dataset's quality, facilitating effective model training and subsequent evaluation, contributing to a comprehensive assessment of the model's performance and capabilities.

5 Implementation

The implementation was conducted in a Python environment, organizing the code across multiple files. The ***build.py*** file houses the main execution logic and all functions required to initiate the input decoding process. Additionally, the ***yolo.py*** file manages the training of the YOLO model if it hasn't been trained already. Lastly, the ***xml_utils.py*** file specializes in reading and parsing XML files.

build.py

The script first of all inspects the input file. If the file path concludes with '*.xml*', the script employs *xml_utils.parse_xml_file(file_path)* method to conduct the XML parsing procedure. In contrast, when the file extension is '*.png*', '*.jpg*', or '*.jpeg*', the script triggers an image classification operation via YOLOv5 utilizing the ***detect.py*** script. Detection is accomplished through refining the confidence parameter, which sets a minimum confidence threshold for object detection predictions, and customizing the IOU threshold parameter, which measures the overlap between two bounding boxes in object detection, based on specific requirements.

After analyzing the input, the system identified the necessary machines to construct the system. This identification was made possible by the *associate_images_to_services* function, which reads the *conf.json* file. *conf.json* contains instructions to pair each component with a pre-existing Docker image stored on DockerHub[8].

Subsequently, all detected services are matched with Docker images.
The *generate_docker_compose* function dynamically generates the *docker-compose.yaml* file, crucial for initiating all Docker containers simultaneously. The web server is associated with port 8080, serving as the potential entry point for system attackers upon container startup. A dummy website was created specifically to simulate the entry point.

Following this procedure, the creation of the Terraform plan becomes essential to orchestrate infrastructure provisioning. The Terraform plan is an essential step because it enables meticulous preparation and evaluation before implementing changes to the infrastructure. This plan serves as a strategic guide outlining the actions that Terraform will execute to achieve the desired infrastructure configuration specified in the Terraform

[8] hub.docker.com

configuration files.

Creating a Terraform plan initiates a dry-run, allowing for a preview of the intended modifications without directly applying them to the infrastructure. This preview enables thorough scrutiny and validation of the proposed changes, ensuring they align precisely with the envisioned alterations. By conducting this preemptive review, any potential issues, conflicts, or discrepancies can be identified and addressed before executing the actual modifications. Ultimately, the Terraform plan mitigates risks associated with unintended disruptions or misconfigurations in the existing infrastructure.

To define the specifics, the *generate_terraform_config_from_docker_compose* function dynamically generates the *main.tf* file using *terraform_template.tf.j2* as a template.

Finally, the Terraform plan is executed to conclude the process.

yolo.py

In the **yolo.py** file, the entire segment concerning the download and training of the model for image recognition of the provided input is implemented. The chosen model is YOLOv5, which is a state-of-the-art object detection algorithm.

The dataset utilized is downloaded in the YOLOv5 PyTorch format and then imported within the code. Through a command executed from the command line within the code, the training of the network is initiated after appropriately configuring the data. The training process was conducted for 23 epochs to comprehensively train the model. This choice was made after observing an increase in loss from epoch 24 onward when experimenting with training for 100 epochs.

xml_utils.py

In this specific file, there is a function named *parse_xml_file* designed to handle the input data in XML format. This function performs checks within the 'cell_style' or 'cell_value' parameters to identify network component names. Based on the identified names (e.g., 'laptop'), these components are assigned a corresponding detection label (e.g., 'Client; detected: laptop').

6 Docker Images

The Docker images employed in this project are bespoke, predefined images tailored to correspond to distinct system components. These images are linked to their respective components via a JSON configuration file (*conf.json*), allowing swift modifications to the associated images through a DockerHub link. Such an approach facilitates the effortless switch between existing images or the creation of new ones, tailored precisely for the specific use case.

For instance, the project opted to create straightforward standard images utilizing Dockerfiles through command-line instructions. These Dockerfiles enable the construction of Docker images, each catering to different component functionalities:

- **MySQL Image Dockerfile:** Sets up a MySQL database with a defined root user password and initialization scripts.
- **Ubuntu SSH Client Image Dockerfile:** Installs client software, like an SSH client, on an Ubuntu base image.
- **Ubuntu Firewall Configuration Image Dockerfile:** Installs and configures a firewall (e.g., iptables) within an Ubuntu environment.
- **Ubuntu Server Configuration Image Dockerfile:** Installs and configures a server application (e.g., Apache) in an Ubuntu environment.
- **Nginx Web Server Configuration Image Dockerfile:** Configures a web server (e.g., Nginx) and copies an HTML file to the server, exposing it on port 8080.

These Dockerfiles serve as the basis for constructing Docker images, each tailored to meet the specific requirements of individual components within the system. They allow for easy modifications or customizations to accommodate the specific needs or configurations of each component.

7 Experiments and Results

7.1 Experiments

The core challenges in this project centered on creating a suitable dataset and training the YOLO network. Initially, a classification-based dataset was employed to meet project requirements. However, the project's primary objective was to precisely ascertain the exact count of components depicted in network diagrams from images.

The experimentation with the classification dataset aimed to identify objects assuming the presence of a single object within each image. However, this approach led to inaccuracies owing to the fundamental nature of classification datasets, which typically categorize images into distinct classes without specifying precise object counts.

Due to the classification datasets' inherent limitation of categorizing images into predefined classes, they lack the granularity required to accurately delineate individual object quantities within an image. Consequently, training the model using this dataset resulted in incorrect estimations as it attempted to predict the presence of a single object within an image. While this method may lead to a more lightweight model, it significantly compromises accuracy, deviating from the project's primary goal of precisely counting components in network diagrams.Therefore, a subsequent decision was made to utilize an object detection dataset, better aligned with the project's objectives.

7.2 Results

The implemented YOLO system operates effectively; however, to enhance its robustness, precision, and adaptability across various image patterns, expanding the dataset with additional images is imperative. The project successfully performs detection tasks, whether using images processed with YOLO or XML files as inputs. Notably, it accurately identifies the number of machines within the image shown in Figure 2, chosen for experiments.

Additionally, it adeptly generates configuration files and links corresponding Docker images, ultimately producing the Terraform plan.

8 Discussion

The primary project goal was to employ defensive deception tactics by creating deceptive services and components resembling high-value targets, diverting potential attackers from the real system. This involved constructing an intricate "fake" infrastructure based on an architectural blueprint.

To achieve this, the project had specific criteria. It aimed to recognize component types and quantities within a given architectural diagram, associating docker images with each component through configurable methods. The project supported two approaches: using computer vision techniques on standard images, potentially leveraging pre-existing models like YOLOv5, and parsing draw.io diagrams via XML export. Additionally, it aimed to generate a terraform plan, utilizing provided docker images as a basis for each resource in the intended architecture.

While the developed system fulfills the specified constraints, it does have areas for improvement. Despite meticulous efforts, the YOLOv5 model's precision posed limitations, especially in accurately identifying machine type and quantity within images. These limitations stemmed not only from the model but also from inherent constraints within YOLOv5 itself.

Additionally, the network's prediction precision was influenced by dataset constraints. Challenges in obtaining a diverse set of online images within time constraints hindered the dataset's balance. Expanding the dataset could certainly enhance performance in this regard.

8.1 Future Work

The project's evolution offers numerous avenues for improving accuracy and usability in identifying architectural components.

A pivotal focus involves enhancing the precision of the YOLOv5 model through extensive dataset training, potentially refining detection parameters and overall performance.

Diversifying input images via an expanded dataset could effectively showcase the network's capabilities while uncovering fresh areas for improvement.

Exploration of advanced computer vision and deep learning methods beyond YOLOv5 stands to substantially enhance the project's accuracy in identifying architectural components. Embedding the network within a PyTorch model or similar frameworks might streamline training and inference processes.

Critical to the project's advancement is the improvement of automation in Docker image creation and Dockerfile generation. Developing a more adaptable infrastructure could bolster the system's agility in generating specific Docker images.

Scalability remains paramount. Enhancing the project's capability to handle intricate architectural diagrams and deploy larger infrastructures would mark significant progress.

Integrating a user-friendly interface could streamline the diagram input process, offering an intuitive platform for user interaction, thereby eliminating the need for command-line execution.

These proposed pathways hold substantial potential for advancing the project's accuracy, usability, and scalability in recognizing architectural components.

References

The code is available at the following link:
https://github.com/alessandrablasioli/fakeorchestrator

[1] **Yolov5 Guide.** (Jacob Solawetz) https://blog.roboflow.com/yolov5-improvements-and-evaluation/

[2] **Deploy a model on a Custom Dataset.** (Jacob Solawetz, Joseph Nelson) https://blog.roboflow.com/how-to-train-yolov5-on-a-custom-dataset/

[3] **Terraform.** (Wikipedia) https://en.wikipedia.org/wiki/Terraform_(software)

[4] **Deception technology.** (Wikipedia) https://en.wikipedia.org/wiki/Deception_technology

[5] **Deception for Cyber Defence: Challenges and Opportunities** (David Liebowitz, Surya Nepal, Kristen Moore, Cody J. Christopher, Salil S. Kanhere, David Nguyen, Roelien C. Timmer, Michael Longland, Keerth Rathakumar) https://www.researchgate.net/publication/362706211_Deception_for_Cyber_Defence_Challenges_and_Opportunities