Fake Architecture Orchestration
Cybersecurity Course Project

Kankana Ghosh
Master's Degree in Artificial Intelligence, University of Bologna
kankana.ghosh@studio.unibo.it

Abstract

Defensive disruption has recently been proposed as an important tool for cyber defense leveraging the existing security measures. It makes attackers job harder because it does more than just block access; by creating fake services and components that appear as valuable targets to attackers, it impacts the decision making causing the attacker to waste time and effort as well as expose their presence in the network. One of the factors limiting the use of disruption has been the cost of generating realistic architecture manually. Recent advances in automation and coding tools have, however, created opportunities first scalable, automated generation of realistic deceptions. This paper describes a Fake Architecture Orchestrator using Terraform and containerized Docker environments for cyber security deception, dynamically generating infrastructure based on draw.io diagrams and custom XML parsing. This automates deployment, streamlines resource management and misleads attackers through strategic deception.

1 Introduction

The digital revolution has had a huge impact in every aspect of life, precision health, digital agriculture, autonomous vehicles, digital government, to mention a few. But as goes the saying, there is no such thing as free lunch – among various other challenges, cybersecurity stands front and its risks pose vast threat with the continued growth of the internet- enabled world (David Liebowitz, 15 Aug 2022).

Achieving a perfect cyber security and building a system secure against all attackers is an ill posed problem and likely impossible. While relentless pursuit of this ideal remains crucial, it's equally important to develop techniques that enable systems to adapt and defend against attackers who breach existing security measures. Deception for cyber defense starts to get towards that goal—to rebalance the asymmetric nature of computer defense by increasing attacker workload while decreasing that of the defender.

Kevin Mitnick, also known as the world's most infamous hacker in his book The Art of Deception, asserted that the human element is security's weakest link. By attacking this link through various deception based social engineering techniques such as pretexting and phishing, cyber criminals have achieved wide success. Deception aims to manipulate humans' perception by exploiting their psychological vulnerabilities, which has direct impact on their beliefs, decisions, and actions. It can be a powerful tool for both hackers and cyber defenders (Li Zhang, 2021). Its potential lies in going beyond mere access blocking, forcing attackers to waste valuable time and resources. Moreover, deception can instill false beliefs in attackers, leading to outcomes beyond what static defenses can achieve.

In the cyber realm, attackers operate solely on information gathered from the target network. The intruder is often thousands of miles away from the network to which he or she is attempting to gain entry, they often rely on unintentionally exposed network details. However, network owners can leverage this by revealing information that they want the attackers to know by strategically revealing deceptive information. The complexity and ambiguity of networks provide a natural environment to add deception.

This project devises a basic infrastructure orchestration utilizing a fake architecture diagram, to deploy a deceptive architecture that acts as a shield on the actual architecture, in order to divert the attacker's attention and resources away from critical assets.

2 Background

Clifford Stoll in the 1980s, had set up an imaginary computer environment (now known as honeypot),

created a fictitious account along with fake documents with enticing names, to lure a hacker and his objectives. The ploy was a success and led to the arrest of the hacker. Stoll published his experience and expanded it to an engaging book, "The Cuckoo's Egg". During World War II, information about upcoming Normandy landings on D-Day was protected by a series of deceptions, Patton's false army in Britain, convincing Hitler that the invasion was taking place elsewhere, all illustrates that the ultimate target of deception is the adversary's mind (Ormrod, 6-8 October 2014).

Attackers can always gain knowledge about a target system or network through a variety of reconnaissance and discovery tactics, while defenders are usually short of intelligence about their adversaries. Such asymmetric disadvantage for cyber defenders is well promised to be re-balanced through the use of defensive deception, which is expected to deliver a game-changing impact on how threats are faced.

The perimeter-based defense strategy utilizing conventional security measures such as firewalls, authentication controls, and intrusion prevention systems (IPS) has been proven feeble against infiltration. Even with the defense-in-depth strategy U.S. Department of Homeland Security (2016), where multiple layers of the conventional security controls are placed throughout the target network, cyber defenders still find it hard to prevent and detect sophisticated attacks like Advanced Persistent Threat (APT) based intrusions. Such targeted attacks typically exploit zero-day vulnerabilities to establish footholds on the target network and leave very few traces of their malicious activities behind for detection. Defensive deception, featured by its capability of detecting zero-day vulnerabilities and its low false alarm rates due to a clear line between legitimate user activities and malicious interactions, can act as an additional layer of defense to mitigate the issues (Li Zhang, 2021).

Decoy systems differ from honeypot technology (Kimberly Ferguson-Walter, 2017). Traditionally, the main purposes of a honeypot are to draw an attacker away from the true network and gather information about the attacker and the threats it poses. Decoy systems tend to be embedded within the true network; and while they can also capture some information (but less than high-interaction honeypots) about attackers who trigger them, this capture of information is not their primary purpose. Their primary purpose is to use decoys to obfuscate the network and confuse the attacker about the true network topology. Attackers are known to recon networks to gather an understanding of the infrastructure so as to be better prepared should they want to perform a specific attack in the future.

Creating realistic deceptive IT artefacts automatically and at scale remains difficult. But with the introduction of various automation technologies, it has become more feasible. The next sections describe the development of an automated infrastructure orchestrator with the help of platforms as services like Terraform, Docker, enabling handling of tedious setup defined by code.

3 System description

Deploying various services on servers requires an administrator to manage the servers. Such an administrator (system administrator) takes care of the servers as such, whether their hardware or the operating system on which the services are running. When managing services, he communicates with specialists for these services (database administrators, web administrators, network specialists, ...). With their help, the system administrator must ensure the compliance of all com-ponents and their proper functioning. With the arrival of virtualization one physical server can serve multiple virtual instances that are transferable to other physical devices without loss of functionality. Docker comes as a solution to all that replacing the costly traditional virtual machines.

Developing a software service requires a strict software development life cycle and process. This process demands controlling all application code through source control management as well as a rigorous versioning and branching strategy. Software services must be deployed to a target run time environment and provisioning that environment through manual user actions is tedious and error-prone. Provisioning manually also becomes prohibitive as the number of resources grow and spread globally over multiple regions. Terraform provides a platform allowing infrastructure resources to be defined in code. This code allows the automation of the infrastructure provisioning and also allows for a strict development and review life cycle, same as the application software.

This project aims to automate the infrastructure provisioning process for complex architectures defined in draw.io diagrams. It leverages custom

XML parsing, Docker containerization, and Terraform scripting to achieve streamlined and efficient infrastructure deployment.

3.1 XML Parsing

draw.io, a versatile diagramming tool, allows users to create various diagrams and export them in various formats, including XML. This XML format represents the entire diagram structure, including elements like shapes, connectors, text labels, and styles. Parsing XML involves reading the structure and extracting data based on defined tags and attributes. The project begins with the ingestion and analysis of draw.io architecture diagrams, a common visual representation of system components and their interconnections. To achieve this, a custom XML parsing solution was developed, enabling the dynamic extraction of relevant information from the diagrams This parsed data becomes the blueprint for generating the deceptive infrastructure. This dynamic approach allows us to adapt the infrastructure based on different diagrams, eliminating the need for static configurations and increasing its adaptability.

3.2 Docker

The demand and the advancement of Linux containers can be seen in the last few years. Docker has become popular very quickly, because of the benefits provided by docker container. Docker containers are widely used to provision multiple applications over shared physical hosts in a lighter form than a traditional Virtual Machine (VM) platform.

Docker is an open source platform that run applications and makes the process easier to develop, distribute. The applications that are built in the docker are packaged with all the supporting dependencies into a standard form called a container. These containers keep running in an isolated way on top of the operating system's kernel.

The docker server gets the request from the docker client and then process it accordingly. The complete RESTful (Representational state transfer) API and a command line client binary are shipped by docker. Docker packages all the files which need for application, library, middleware, OS, network configuration, etc. into a quantitative form called a Docker image. The Docker image is uploaded to a remote repository called Docker image repository used to share the Docker images among users. Docker constructs a container environment based on its Docker images. Containers hold the whole kit required for an application, so the application can be run in an isolated way. For example, suppose there is an image of Ubuntu OS with SQL server, when this image is run with docker run command, then a container will be created and SQL server will be running on Ubuntu OS.

Docker plays a pivotal role in the project by providing a standardized and isolated environment for deploying fake infrastructure components. Each component identified from the architecture diagram is associated with a specific Docker image. This strategic mapping ensures the accurate representation of resources and simplifies the deployment process. By utilizing pre-built containerized environments for each component.

3.3 Terraform

Terraform is an Infrastructure As Code (IaC) client tool developed by HashiCorp. It allows the user to define both cloud and on-premise compute resources in human-readable configuration files. These files are created using the HashiCorp Configuration Language (HCL). The syntax is declarative with each block of code defining a resource to be provisioned. Declarative definitions (versus imperative) allow the user to define the desired state, rather than an exhaustive list of all the interim steps required to achieve that state.

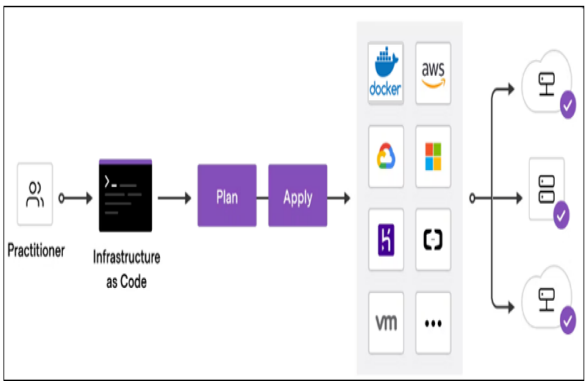

Figure 1: Terraform Workflow

The workflow starts with the generation of the resource configuration code that defines the providers and services required. Multiple providers may be included. The code is contained in configuration files with a .tf extension. The parsed information and Docker image mappings are used to automatically generate Terraform resource configurations. Terraform's declarative language ensures a clear and error-free representation of the desired infrastructure, simplifying deployment and management

(Howard, 21 May 2022).

The configuration file are automatically generated in the HashiCorp Configuration Language (HCL). The code is written in blocks with each block representing an infrastructure object. A Terraform configuration is a complete document in HCL telling Terraform how to manage a given collection of infrastructure resources. Figure 3 shows example code that declares the required provider plugin; Docker in this case. A provider defines what provider plugin is used to translate the resource block into API calls to the infrastructure provider. The resource block is used to define a concrete resource in the provider's infrastructure. Resource blocks are translated into create, update or delete API calls to the provider's target infrastructure service.

Next the Terraform tool need to generate the plan. Running the Terraform plan scans local directories for generated resource configuration files ending in .tf and processes these into a list of actions to be sent to the provider(s). This list is the execution plan that includes all create, update and destroy actions needed to make the target infrastructure match what is declared in the configuration code. The final stage in the Terraform workflow is to apply. executes the actions proposed in a Terraform plan against the corresponding Docker provider. Figure 1 shows the Terraform workflow

4 Experimental setup and results

4.1 Component Detection

A custom made draw.io diagram of a fake architecture was used as the input for component detection. This diagram as shown in Figure 2 visually represented the desired fake infrastructure with various components and connections. The custom XML parser extracts component information, including names, types, and any associated attributes, from the draw.io diagram. No predefined component schemas are used, allowing for flexibility and adaptability to various architectures. The Python code utilizes the *xml.etree.ElementTree* module to extract relevant information from the XML data. Each component is represented by an ID and a value.

4.2 Docker Image Mapping

The parsed information is used to map detected components to specific Docker images. This mapping is achieved by loading a configuration file

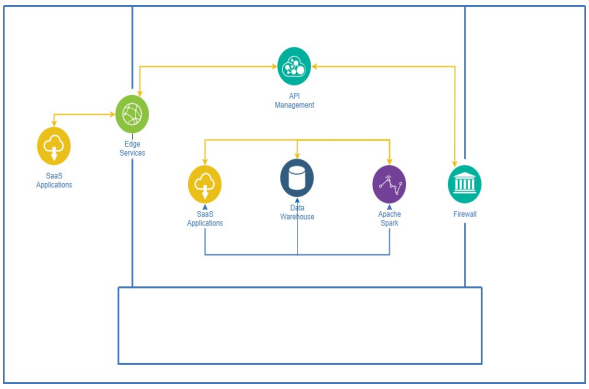

Figure 2: The fake architecture diagram generated from draw.io

(config.json), which associates each component type with a corresponding Docker image configuration, aligning each architecture component with a containerized environment. The system leverages pre-built containerized environments for each component, enhancing efficiency and reducing deployment complexities. Docker plays a pivotal role in ensuring accurate representation and resource utilization during deployment. To avoid duplicate image deployments, only the first encountered instance of a specific component type had its Docker image appended to the output list. Returns a list of detected components with their associated Docker images. The configuration file is manually maintained to ensure accurate component-to-image mappings.

4.3 Terraform Plan Generation

The extracted component information and Docker image mappings are used to automatically generate Terraform resource configurations *(main.tf)* that includes resource blocks for each detected component. The configuration specifies the necessary attributes and configurations, utilizing Terraform's declarative language HashiCorp Configuration Language (HCL). Terraform's declarative language ensures a clear and concise representation of the desired infrastructure, reducing manual effort and potential errors. Generated Terraform code can be easily reviewed, modified, and version controlled for better infrastructure management. The Terraform configuration *'main.tf'* includes provider specifications for Docker and the detected components as resources with separate resource block for the latest precise Docker Image and its corresponding Container for each component, ensuring seamless integration as shown in Figure 3. The Terraform

Figure 3: A snapshot of the HCL code declaring the providers and resources

tool is run through the local Command Line Interface (CLI). The Terraform working directory houses the configuration file generated to describe the infrastructure that Terraform will create and manage. To create a new configuration we need to initialize the directory with *terraform init*. Initializing a configuration directory downloads and installs the providers defined in the configuration, which in this case is the docker provider. The *terraform init* command prints out which version of the provider was installed. terraform also creates a lock file named *.terraform.lock.hcl* which specifies the exact provider versions used. Next running the *terraform plan* command that generates a plan that includes all create, update and destroy actions needed to make the target infrastructure match what is declared in the configuration code and acts as a checkup step for any correction and changes before finally deploying the architecture. Finally running the *terraform apply* command on executes each action listed in the previously generated plan. All the applied configuration data are wrote to the *terraform.tfstate* file, storing the IDs and properties of the resources it manages in this file, so that it can update or destroy those resources going forward and keep track which resources it manages.

In summary, the system seamlessly integrates XML parsing, Docker containerization, and Terraform orchestration to dynamically generate and deploy fake architectures. The custom XML parsing module, Docker image mapping, and Terraform configuration generation collectively contribute to achieving the project's goal of efficiently orchestrating a complex "fake" infrastructure from architectural diagrams. The experiment resulted in the detection of six unique component types within the draw.io diagram as in Figure 2. The Terraform plan encompasses six resource blocks corresponding to these components, leveraging six unique Docker images for deployment. The generated plan passed basic validation, indicating its potential for infrastructure creation.

5 Discussion

The project successfully demonstrated its core functionality in parsing a draw.io diagram, accurately identifying and mapping the components identified and associating them with specific Docker images. Utilizing Docker for containerization and Terraform for infrastructure orchestration streamlined the deployment process. The generated Terraform plan served as a foundation for deploying the fake infrastructure, showcasing the automation potential. The project's modular design could be extended to support various diagram formats and integrate with different deployment platforms, expanding its applicability.

The current implementation focuses on the core mechanics, lacking advanced configurations and broader diagram format support, including specific ports, volumes, and environment variables to tailor container behavior. As the project relies on local machines for deployment using Docker, limitations in storage space and time may arise. Integrating the project with cloud platforms like Terraform Cloud, AWS, Azure, or GCP could overcome limitations of local machines regarding storage space and processing power. This would enable scaling the fake infrastructure deployment and management to potentially larger and more complex scenarios.

6 Conclusion

Despite its current stage, the Fake Architecture Orchestrator exhibits promising potential as a defensive deception tool. By creating believable and

dynamic fake infrastructure, it can help deceive attackers, divert their attention, and protect critical assets. Integrating it with cloud platforms would further enhance its capabilities and impact in real-world security scenarios. Its current limitations provide exciting opportunities for future research and development. As the cybersecurity landscape continues to evolve, this project lays the groundwork for innovative defensive strategies that can adapt and respond to the dynamic nature of cybercrimes.

References

Kristen Moore Cody J. Christopher Salil S. Kanhere David Nguyen Roelien C. Timmer Michael Longland Keerth Rathakumar David Liebowitz, Surya Nepal. 15 Aug 2022. Deception for cyber defence: Challenges and opportunities.

Michael Howard. 21 May 2022. Terraform — automating infrastructure as a service. arXiv:2205.10676v1 [cs.SE].

Temmie Shade Kimberly Ferguson-Walter, Dana Lafon. 2017. Friend or faux: Deception for cyber defense. volume 16, pages 28–42.

Vrizlynn.L.L. Thing Li Zhang. 2021. Three decades of deception techniques in active cyber defense - retrospect and outlook. *Computers Security*, 106.

D Ormrod. 6-8 October 2014. 'the coordination of cyber and kinetic deception for operational effect: attacking the c4isr interface'. In *Proceedings of the 2014 IEEE Military Communications Conference*, pages 117–22.