Cyber-Threat Emulation

For Submarine-Board Containerized Applications

Naval Undersea Warfare Center

NUWC WSU COUGS

Hunter McClure

Stephen Emmons

Evan Kimmerlein

Table of Contents

I. Introduction

Security and modularity requirements for the software onboard submarines operated by the US Navy mean much of the software in use on these vessels is containerized, or in other words packaged just with the libraries, OS, and other dependencies required to run the software. The NUWC has recognized a need for testing the security of these systems and has tasked us with emulating them and testing how they might perform under attack from malicious actors. The ultimate end goal of this project is to create a representation of containerized software that may be found on submarines using Kubernetes, and to develop additional overlaid software that will emulate cyber-threats within the system we've created, as well as evaluate the impacts of the cyber-threats to the containerized system.

I.1. Background and Related Work

The domain for this project is firmly in the cybersecurity field, where we expect the primary challenge of this work to come from implementation of cyber threat emulation and the evaluation of the effects of cyber attacks carried out on our containerized software. The Defense Technical Information Center has compiled an overview of useful cyber-threat modeling techniques, as well as an original framework for modeling proposed in the paper. On page 55 specifically, it stresses the need for a threat model to represent adversary activities, whether that be protection from data modification, degradation, fabrication, or interruption at the network layer, and it must be able to detect when these things are taking place as well as what systems are affected[2]. Our model will need to be capable of implementing all of these key aspects, as well as adding the vocabulary necessary to our model specific to a submarine context.

While the main structure of our containerized software will be ultimately dictated by what naval submarines already use and support, IEEE has published a framework for mission critical containerized systems that may be useful, especially in a military context as high-data availability and integrity are most critical. The paper, titled "Managed Containers: A Framework for Resilient Containerized Mission Critical Systems" explains the concept of managed containers involving 6 layers, which give the system reduced overhead and increased modularity[1]. IEEE and this paper in particular, coined the term managed containers[1]. We expect our emulated system to use some form of application layering, as it is an effective strategy for dealing with the most common forms of attacks, known as container escaping, inner-container attacks, and cross-container attacks, all of which are explained at the end of Section II.

I.2. Project Overview

Cybersecurity has become a NEED instead of a want over these last couple of years. This is due to the fact that many hacker groups have started to come out of the shadows such as the infrastructure pipeline attack on the American east coast[4] and the recent media Rockstar games/Airbnb attack. Now, with hostile countries starting to use cyber attacks as a key playing card on the battlefield, our government is in need of help, and thus the NUWC (Naval Undersea Warfare Center) has asked us for help. The NUWC has stated that there is a need to understand risks and develop threat management strategies for new software architectures that are being put in place in submarines, which are container-based rather than virtual-machine-based. Furthermore, we have been asked to collaborate with the Naval Undersea Warfare Center to develop techniques needed for cyber-threat analysis and mitigation.

The team at Washington State University will be working with a team of experienced computing professionals from the sponsor of NUWC. We aim for a positive work environment and gain great insight from the experienced individuals while working on the emulator. This knowledge will help the American Navy as well as us college students as we get close to graduating and heading into careers.

In the end of the project, we aim to have a better understanding of cyber-security issues arising in containerized software systems, including especially for the special submarine environment. The team also wants to be successful with using the Kubernetes system as a way of orchestrating containerized software. We aim for software that emulates cyber threats and to progress further into the field of cybersecurity during these technologically evolving times.

This emulating software will hopefully help the larger need for cyber security in the United States military as we continue to learn more about cyber threats so that we may be ready for attacks.

I.3. Client and Stakeholder Identification and Preferences

Our primary client is Ben Drozdenko and the Naval Undersea Warfare Center here in the United States. The product of the emulating software will hopefully be used in the Naval branch as a way to help increase our knowledge in cybersecurity and specifically cyber threats.

We will be working with a team from NUWC and hope to build successful software that will be used for threat-analysis and mitigation. The goal is to plan with NUWC on how to go about the creation and use of this technology and then execute this plan perfectly so that all requirements of NUWC are met.

II. Team Members

Hunter McClure is the team lead responsible for communication with our clients and professor, as well as ensuring group cohesion and efficiency. Hunter has also worked on the Github repo and documentation. He is a computer science major interested in cybersecurity, and has previously worked on developing databases and websites used to connect people with resources such as ride shares and ballot drop boxes among other things. His skills include Python, Java, SQL databases, and project management.

Stephen Emmons is a computer science major interested in algorithm and database design as well as information technology. For this project he has worked jointly with other group members on all aspects, responsible for much of the documentation as well as ensuring system functionality and viability. He currently works as an IT Technician for the Carson College of Business within WSU, and is skilled in Python and Java, as well as management of information technology especially in regards to VMs and Windows systems.

Evan Kimmerlain is a computer science major interested in machine learning and neural nets. For this project he is primarily responsible for documentation and the maintenance of the containerized system. Evan also initially made the container and created a vdi image so others may access it. He has previously developed a cancer detection neural net. His skills include, machine learning, C++, Java, Python, and OS management and development.

III. System Requirements and Specifications

III.1. Use Cases

Name	Containerized Software Emulation
Users	Developers
Rationale	In order to properly learn how to apply cyber threat analysis to containerized systems a software that emulates these systems is required.

Triggers	Whenever a new functionality or system requirement is conceptualized, developers will use kubernetes to implement a high level version of it.
Preconditions	A kubernetes architecture ready to accept new containers.
Postconditions	A new container is deployed to kubernetes with required functionalities.
Acceptance Tests	The container is able to successfully run in kubernetes with no issues.

Name	Cyber Event Analysis and Testing
Users	Researchers/Developers
Rationale	Cyber threat testing and analysis will be a core functionality of our work on this project. In order to do this there will need to be software that can accommodate such analysis in regards to containerized systems, both by deploying said threats and providing means to observe their impact.
Triggers	The user will launch an attack either through a console command or button press.
Preconditions	A functional container is present in Kubernetes and target(s) is selected.
Postconditions	The attack is carried out, and a report of its effects is printed.
Acceptance Tests	The effect on the containerized system can be observed and reported statistics are verified as accurate.

Name	Cyber Threat Emulation
Users	Developers
Rationale	In order to properly evaluate the effects of cyber attacks there must also be an emulation of said attacks that can be deployed to the test software.
Triggers	When the need for analysis of a new cyber threat arises, a script will be created to interact with the program maliciously.

Preconditions	Containers have been developed and deployed to the Kubernetes architecture.
Postconditions	The script can successfully interfere or attempt to interfere with the operation of the targeted containers.
Acceptance Tests	The script has an observable impact on the operation of the targeted containers within desired parameters, or otherwise affects the system in an abnormal way that can be observed and understood.

III.2. Functional Requirements

III.2.1. Risk Analysis

Quantitative Risk Analysis: The application must have the capability of quantitatively representing potential risks within a containerized environment. This report should be easily understandable and cover most known vulnerabilities.
Source: NUWC Benjamin Drozdenko
Priority: 0 - Essential for functionality

Cyber Threat Simulation: The application must have the ability to simulate various known cyber attacks on the system. These effects of these attacks must be properly tracked and passed on to the Quantitative risk analysis.
Source: NUWC Benjamin Drozdenko
Priority: 0 - Essential for functionality

Physical Threat Simulation: The application must have the ability to simulate various physical attacks upon the system. These effects of these attacks must be properly tracked and passed on to the Quantitative risk analysis.
Source: NUWC Benjamin Drozdenko
Priority: 0 - Essential for functionality

III.2.2. Containerization implementation

Docker and Kubernetes compatibility: The application must utilize Docker and Kubernetes for containerization.

Source: NUWC Benjamin Drozdenko
Priority: 0 - Essential for functionality

Multi-pod Support: The application must have the ability to load a large variety of pods for testing. Testing will include analysis of how these groups of pods interact with each other.
Source: NUWC Benjamin Drozdenko
Priority: 0 - Essential for functionality

III.3. Non-Functional Requirements

Generalistic Approach
A generalist approach is necessary for the development of this software. Many pods and environments that the application will be used to test may not be known or even accessible to the development team. For this reason the development must not lock itself to specific environments and pods.

Cyber Physical
The system shall be effective at showing the cyber physical impact of cyber attacks. The system should show the real physical consequences for vulnerabilities within the container being environment.

Submarine Based Focus
This system shall be focused on submarine based systems. While a generalist approach will be used, this system is intended for use on submarines. For this reason prioritization for proper analysis of container environments that would be found on submarines will be taken.

Proof of Concept
This system shall provide a clear proof of concept, to ensure that it is behaving as anticipated. This may include example environments and outputs to demonstrate its functionality.

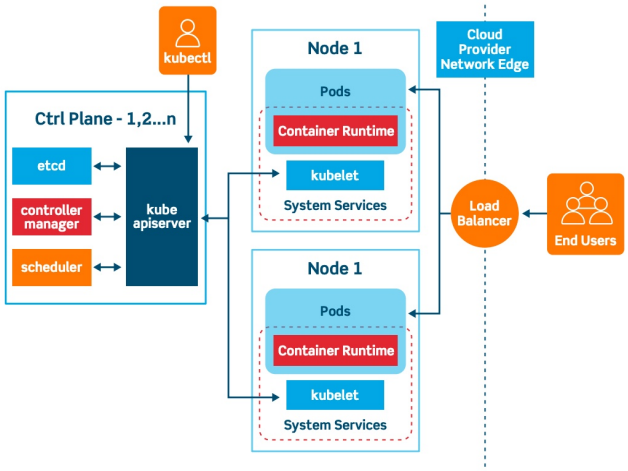
IV. Architecture Design

IV.1 Overview

Our architecture will follow the standard Kubernetes layout, consisting of a Control Node which contains the scheduler and controllers[6]. Through this node we will control the operation of our other nodes, each containing its own container runtime, where each runtime is an abstraction of software one might find on a navy submarine. Whether or not the cyber threats we develop will be a part of the Kubernetes architecture or developed outside of it will be decided later on in the project, but we do know they will have to be able to access the nodes in some way.

The Control Node, as stated before, will schedule the container runtimes executing them as the need arises. It will also maintain the API for the project, allowing containers to communicate with each other if the need arises. Each following container node will have a container as the name suggests, each of these containers[9] being responsible for a certain functionality one might find in a submarine such as fire control or sonar monitoring. These functions will merely be emulated as best they can be with the knowledge and descriptions given to us by the project sponsors, as we do not have access to real life versions of this software. The container nodes will also contain Kubelets[7], which allow the nodes to communicate with the overarching Kubernetes architecture. The containers themselves will be implemented using Docker, which will let us virtualize the server OS the containers will be running on. Below is an example diagram of what a generic Kubernetes application looks like.

IV.2 Subsystem Decomposition



IV.2.1 Control Plane

Description: The Control Plane will manage each other node, allowing for manipulation, and scheduling of runtimes so that they may execute when needed.
Concepts and Algorithms Generated: The Control Plane includes all of what is needed for each Kubernetes pod to work together. It acts as a middle man meaning all worker nodes within the Kubernetes Cluster will have to communicate and operate through it, as opposed to directly with each other.
Interface Description:
Services Provided:

1. *Service Name:* Schedule Runtime
   Service Provided to: BackEnd
   Description: The Control Plane will schedule runtime events for other nodes
2. *Service Name:* API Server
   Service Provided to: BackEnd
   Description: Kubernetes API server module, allows access to functionality of other Control Plane services
3. *Service Name:* Scheduler
   Service Provided to: BackEnd
   Description: Control Plane Scheduler for Worker Node runtimes
4. *Service Name:* Controller Manager
   Service Provided to: BackEnd
   Description: Manages controller processes such as responding to nodes going down, creation of pods, and populating endpoint objects0
5. *Service Name:* Kubectl
   Service Provided to: BackEnd
   Description: Command Line tool for communicating with the Control Plane

Services Required:

1. *Service Name:* Kubelet
   Service Provided from: Data Plane
2. *Service Name:* Kube-Proxy
   Service Provided from: Data Plane

IV.2.2 Data Plane:

Description: The Data Plane is composed of worker nodes. These worker nodes will be comprised of various pods. These pods will differ depending on the packages the system is checking for vulnerabilities.
Concepts and Algorithms Generated: Analysis of the Data Plane, and how each worker node and pod interact is the main focus of this application. Pods must be able to be interchangeable in this system, as to allow for dynamic testing of packages.
Interface Description:
Services Provided:

1. *Service Name:* Kubelet
   Service Provided to: BackEnd
   Description: Kubelets are Node Agents that run each worker node within the Data Plane, ensuring pods are running healthy.

2. *Service Name:* Container Runtime
   Service Provided to: BackEnd
   Description: Installed into each node, container runtimes allow for each nod within the cluster to run Pods
3. *Service Name:* Kube-Proxy
   Service Provided to: BackEnd
   Description: Network proxy that reflects services defined in the Kubernetes API.
4. *Service Name:* Etcdctl
   Service Provided to: BackEnd
   Description: Command Line tool for communicating with the etcd

Services Required:

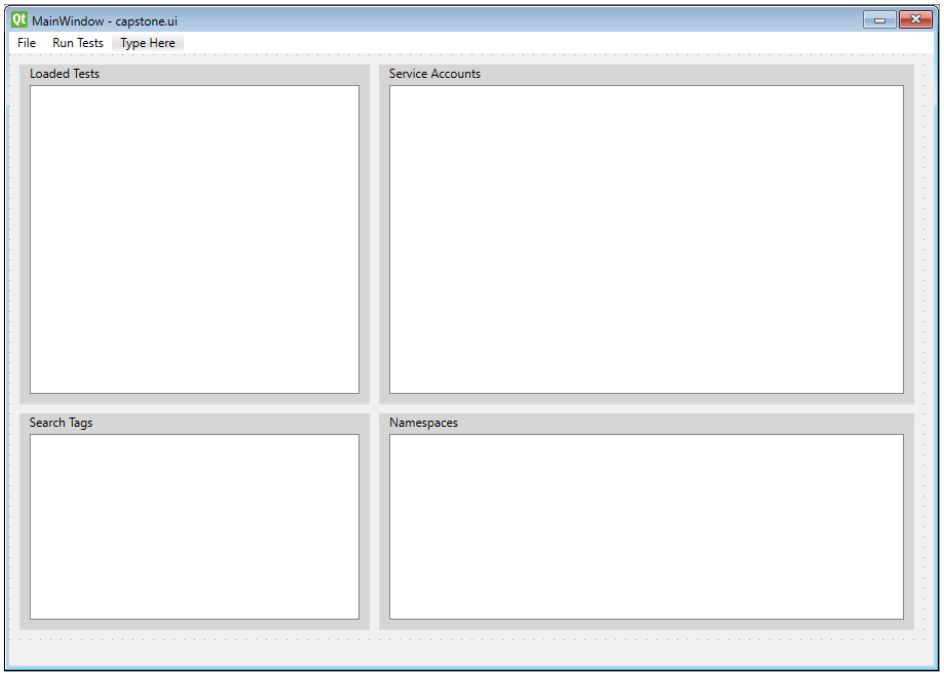
1. *Service Name:* API Server
   Service Provided from: Control Plane

IV.3 Data Design

Given that Kubernetes itself handles much of the data storage we are interacting with on the backend, we will mainly be dealing with standard data structures such as arrays, lists, and hash tables in our development. We did not see a need for a database and therefore did not construct one. In general, cyber threat analysis is performed via execution of shell scripts within the Kubernetes system itself, and our programs are simply responsible for loading and executing the data within these scripts.

IV.4 UI Design

The UI for our application utilizes the Python QT Designer library and application, as it is a simple tool with drag and drop functionalities for creating a clean and easy to use UI. The UI's function is mainly a simple way for users to load and launch test cases for the kubernetes environment, with additional information on relevant metrics from the test displayed. The UI is incomplete as our clients urged us to focus our efforts on furthering the test environment and generation of a demonstration. On the following page is a screenshot of the UI in its current state:



V. Test Plan

V.1. Test Objectives and Schedule

Our team will utilize testing to ensure our application's integrity, functionality, and performance. Although exhaustive testing is impossible, our tests will be written to efficiently cover areas of high stress and potential breaking points in our software. Our testing will also showcase examples of discovering known vulnerabilities, and analysis of their impact as proof of effectiveness of our software.

We will utilize the Kubernetes End-To-End Framework[8] to implement the testing of our system. In situations where E2E fails to meet our requirements we will utilize libraries such as K6 to fill the gaps. What libraries are utilized and how they interact will likely change and be updated as the project progresses.

Our testing will have two deliverables: First, our testing code posted to github. This code will be posted to github, however much of this code will likely not be runnable of the gate due to the nature of working with Kubernetes systems. Proof this code's functionality will be in our next deliverable, testing documentation. Testing documentation will give an overview of our test cases, explaining both our reasoning for the tests we implemented and their results. Tests that show proof of concept of our code, like the ones that reveal known vulnerabilities and analyze

their impact will likely have more in depth documentation and explanation than simple unit tests that ensure the validity of our code. Milestones for our testing will include: writing our code, unit testing the code, integration testing code, system testing our code, and finally proof of concept testing our code. The documentation will be updated as tests are produced. The milestones will likely repeat in a cycle, with earlier milestones repeating more often than the later ones.

V.2. Scope

This section gives an overview of our testing strategy and testing guidelines. It also gives an overview of each testing framework, and how they are utilized. As discussed above, we may include more testing frameworks as the project develops. As such, this section will be updated as new frameworks are added.

1. Testing Strategy

We will provide testing for all non trivial functions within our code. Integration testing will be performed to guarantee proper functionality between functions. In addition higher level testing will be provided to show proof that our software meets the client's given specifications and requirements. This process will follow this basic cycle:

A. **Code Creation:** The developer creates code necessary to complete or aid the completion of a requirement. This will primarily be the code that builds up our subsystems.
B. **Unit Test Creation:** Developers write tests for the newly created code.
C. **Code Testing:** Developers run newly completed tests to ensure proper functionality of new written code.
D. **On Failed Tests:** Debug code, fix bugs and ensure the code is properly functioning, and all tests now pass. If no tests failed, skip this step.
E. **Create Merge Request:** Now that the code has passed its tests, the developer will make a merge request from their current branch. Other developers should be informed of this new request and added as reviewers.
F. **Await Approval:** This merge request must wait for completion until it receives review from another team member.
G. **Resolve Conflicts/Merge:** Now that it has approval, resolve merge conflicts and merge to the parent branch.

NOTE: Although this cycle will be followed for all potential github code, due to the nature of working with Kubernetes and VM environments, this cycle may not be followable for much of our work/testing.

V.3. Test Plans

V.3.1. Unit testing

Unit tests are performed using the standard unittest library in Python, which we use to validate program output as well as the overall functioning of our system. Unit testing in our project is used in particular to test data within the system, for example testing whether Secrets in our Kubernetes environment are encrypted or not, or testing whether secrets are visible to the outside world. This means a failed test does not always mean the program isn't working as intended, rather unit tests are a quick tool to obtain working knowledge of how the system is exposed.

V.3.2. Integration testing

With integration testing, our team will be focusing on the entry points of the container, especially when it comes to cyber threats. A big focus for entry points will be backdoors and brute forcing. We will be exploring and testing threats that will focus on entering the system and then seeing if we can potentially fill those gaps that allow for those threats to get into the system. Furthermore, once we find these bugs and/or threats, we will continue to add to our system and then test again. Our team will be using a top-down strategy when testing the system and the container's subsystems so that we may incorporate all parts of the software. However, we may switch to bottom-up testing if we find that top-down does not work as efficiently as we want it to. Finally, if our clients want more testing/assurance in any part of our system then we would want them to communicate/reach out to us on that subject.

V.3.3. System Testing

V.3.3.1. Functional Testing

We will focus our functional tests around the requirements specified in section II.2 of this document, mostly by obtaining the correct metrics to ensure we have an accurate picture of whether each requirement was met. We expect this will be accomplished mainly through data analysis and analytics obtained from the Kubernetes UI as well as any additional tools we may require or develop. Failure will then be determined based on comparing expected outputs against measured outputs for a requirement, and in the case a failure is detected, an issue will be created in the github repository, assigned based on workload and expertise/knowledge on the issue at hand.

V.3.3.2. Performance Testing

Performance testing will look largely like functional testing, but with a focus more on the health of the system as a whole, where attributes such as system structure, accuracy of data, and operating speed/efficiency will be the primary focus. A special emphasis will be placed on stress testing, where we will attempt to find as many of the system's limitations as possible, and ensure they are known and accounted for. We will especially need to make sure we know under what kind of stress our functional requirements fail in order to ensure the non-functional requirements listed in section II.3 have been met. Just as we will do for functional testing, an issue will be created in the GitHub repository if a member of the team feels the system does not perform as it should based on the metrics reported in our tests.

V.3.3.3. User Acceptance Testing

User acceptance testing will be carried out with our clients, wherein we will give a short demo of our work and explain the intricacies of what we've developed so they can get a clear picture of what we've done and how it fits into the requirements they've set out for us. We expect to do this several times at least, each time taking in the client's feedback and documenting it, then creating an actionable plan to make necessary changes if required, as well as incorporating their feedback into our future efforts so as not to wander down the wrong path should we have made any mistakes during previous sprints. We expect a large portion of user acceptance testing will be educating the clients on how to properly interact with our work, therefore we will need to incorporate accessibility and ease of use into our testing, which will likely be mostly observational in nature.

V.3.3.4. Test Cases

Test 1: Minikube starting test

Aspect:	Ensure that Minikube startup succeeds by checking Minikube status
Expected:	Type set to control plane, Host, kubelet, api server show running Kubeconfig shows Configured
Actual:	Type set to control plane, Host, kubelet, api server show running Kubeconfig shows Configured
Test Result:	Pass
Requirements:	Minikube must be installed

Test 2: Kubectl default services

Aspect:	Ensure that Minikube is showing the expected default services
Expected:	Kubernetes, kube-dns, dashboard-metrics-scraper, and Kubernetes-dashboard should all be present
Actual:	Kubernetes, kube-dns, dashboard-metrics-scraper, and Kubernetes-dashboard are all present
Test Result:	Pass
Requirements:	A fresh instance of Minikube must be started

Test 3: Kubectl default pods

Aspect:	Ensure that Minikube is showing the expected default pods
Expected:	A specific list of names of pods that should be running on Minikube by default.
Actual:	The appropriate specific list of names of pods that should be running on Minikube by default
Test Result:	Pass
Requirements:	A fresh instance of Minikube must be started

Test 4: Kubectl running pods

Aspect:	Ensure that Minikube is running the expected default pods
Expected:	A specific list of names of pods that should be running on Minikube by default and their status as running
Actual:	The appropriate specific list of names of pods that should be running on Minikube by default with a status of running
Test Result:	Pass
Requirements:	A fresh instance of Minikube must be started

Test 5: Kubectl create deployment

Aspect:	Ensure that Kubectl can properly add new deployments
Expected:	A new entry in the active deployments should appear
Actual:	A new entry in the active deployments appeared
Test Result:	Pass
Requirements:	A fresh instance of Minikube must be started

Test 6: Kubectl delete deployment

Aspect:	Ensure that Kubectl can properly add remove deployments
Expected:	The list of active deployments should be empty
Actual:	The list of active deployments were empty
Test Result:	Pass
Requirements:	A fresh instance of Minikube must be started, with only 1 deployment active, named test_deployment

V.4. Environment Requirements

In order to properly interface and interact with the system at hand virtualization software must be installed, with either VirtualBox or KVM2 recommended. Kubectl will also need to be installed, as it will allow the user to interact with the Kubernetes cluster. Docker will also need to be installed, as it will be the application used to create and manage containers. Minikube will also be required, as it will be used to create VMs on the user's local machine and deploy a cluster within it. Finally, we recommend installing all of the previous within a linux OS such as Ubuntu or Kali, as most of the documentation that accompanies the software used for our system is written for Linux users. Finally, the system running the tests must have Python 3 installed and must be able to run shell scripts on the kubernetes system as a whole. In terms of hardware requirements, our programs are not resource intensive, and anything reasonably modern will work.

VI. Projects and Tools Used

At the moment our technologies in use consist of:

- Kubernetes
- Docker
- Minikube
- Virtualbox
- Ubuntu
- Microsoft Threat Modeling Tool
- Qt Designer
- Python 3
- Kubectl
- Etcdctl

VII. Final Prototype Description

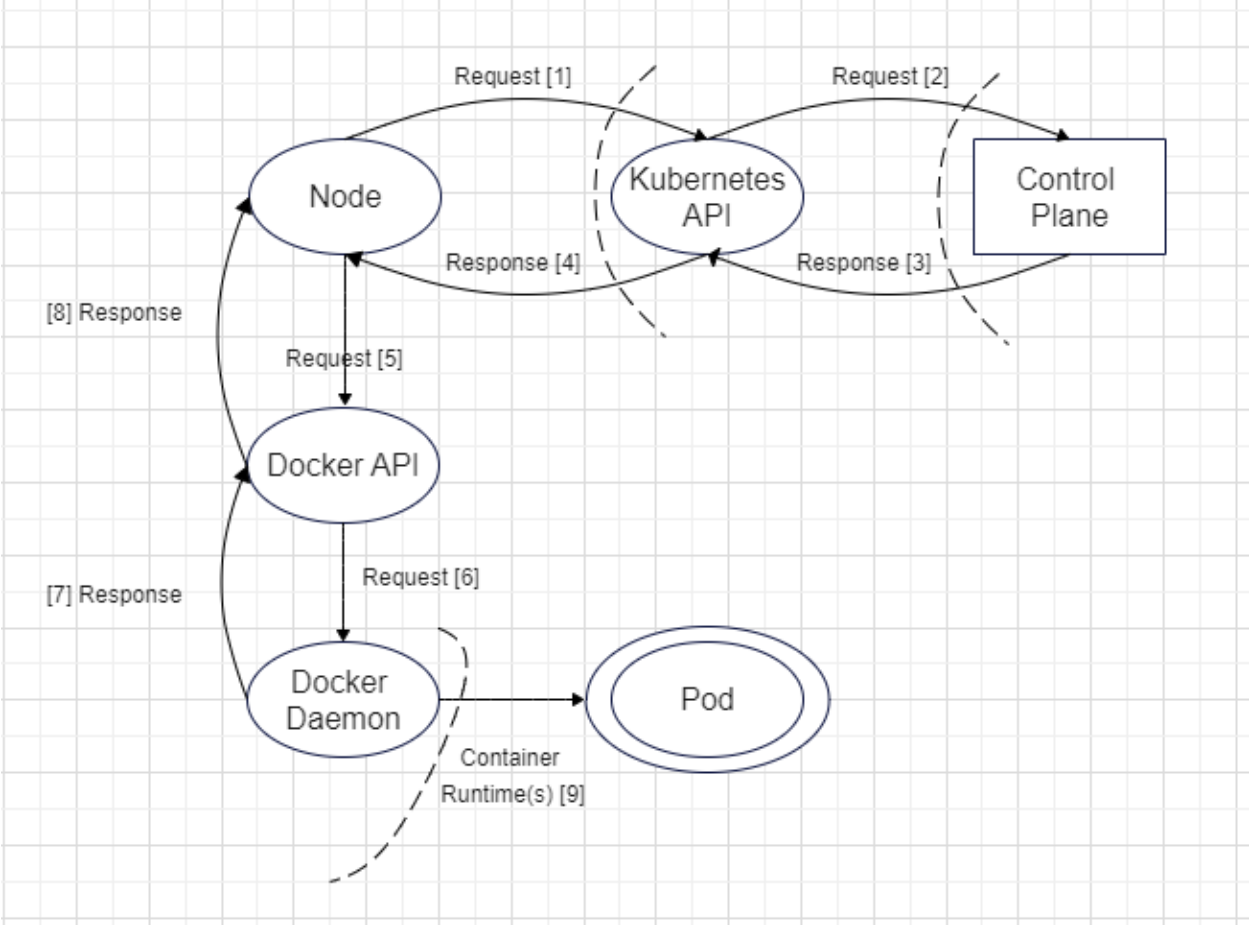
We are working toward a comprehensive testing suite that allows for loading and execution of Kubernetes cyber threats and tests of system vulnerability.

Our final prototype primarily consists of Python programs running in the Ubuntu OS with a Kubernetes cluster installed and operational, as described in previous and later sections of this report. Running the program in the Ubuntu command line will open the UI and allow users to load shell scripts which can perform imitated or emulated cyber attacks on the Kubernetes cluster. The application of these shell scripts is not limited to cyber attack, they can also be used to gain statistical data on the cluster. In essence, what the shell scripts do is up to future users, we have simply developed a few samples to test our system's functionality and software that enables their execution. The primary function of our product is to load shell scripts into the system through our UI, and execute them on the cluster. We have also developed tests that can

be performed to glean useful information about the cluster, such as Secrets visibility and encryption, which can be used and expanded on by future users/developers.

VII.1 High Level System Diagram

The STRIDE diagram below is a high level representation of our virtual machine environment and the way it interacts with Kubernetes and Docker, along with a summary of potential vulnerabilities within the system we have recognized and may attempt to explore. We have not explored or tested each exploit listed, and it is not an exhaustive list.



[1, 2, 3, 4, 5, 6, 7, 8]

- Tamper. - Man in the Middle to read commands and responses
- Info Disc - Net Eavesdropping
- Deny - Network DDos

[Node]

- Spoof- commands to Kubernetes API
- Tamper - malware on node hardware
- Repudiation - forged authentication on node hardware
- Deny - crash node hardware

[Control Plane]

- Spoof - response to API
- Tamper - mitm
- Info Disc - unencrypted secrets
- Elevation - gaps in firewalls, security policy, access control

[Kubernetes API]

- Spoof - request, response
- Tamper - mitm, launch fake or modified containers

[Docker API]

- Spoof - request to Daemon, response to Node
- Spoof - fake pull request
- Spoof - redirect to wrong repository
- Tamper - mitm, launch fake or modified containers

[Docker Daemon]

- Spoof - runtime creation
- Deny - crash, DDos

[9] Container Runtime(s)

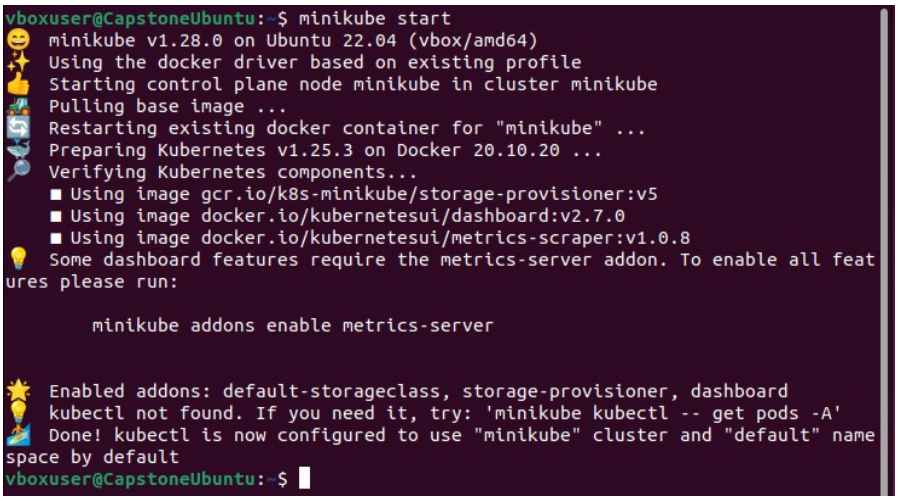
- Tamper - malicious container to pod
- Info Disc - container functions

[Pod]

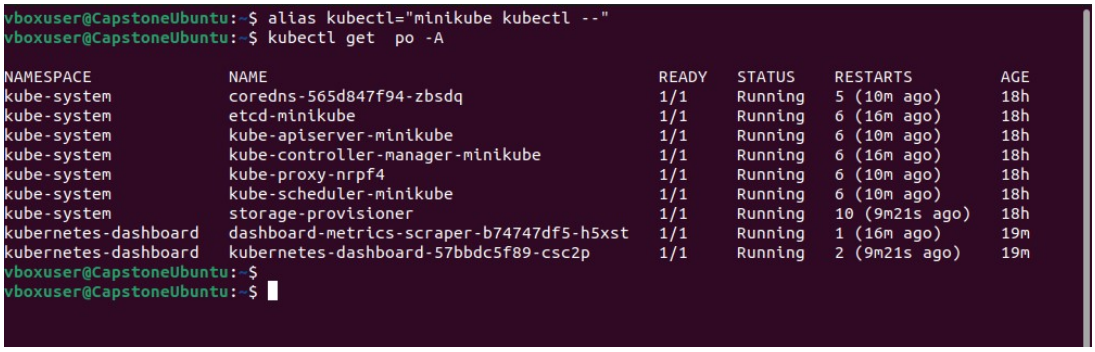
- Repudiation - Misconfigured security policies

VII.2 Minikube Implementation

Sections VII.2 and VII.3 demonstrate how we set up our virtual machine environment. Minikube is used to set up and manage local Kubernetes clusters for development without interfering with deployed applications. Attached below is a screenshot confirming successful installation.

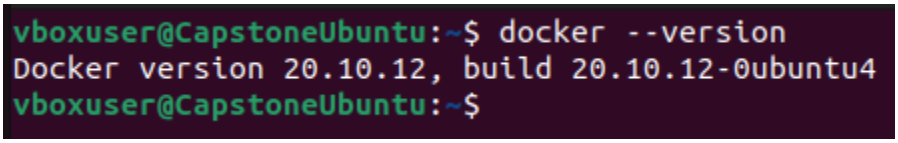


In order to utilize Minikube properly we also needed to install kubectl to interface with the cluster control plane through the command line. Below is a screenshot confirming kubectl installation through Minikube.



VII.3 Docker and Kubernetes Implementation

As Minikube uses docker to manage containers and virtualize applications, it was important we get it installed initially. Below is a screenshot confirming installation that details the version we are working with.



VII.4 Containerized Installation and setup scripts

The previously described environment can be loaded via a virtualbox image that guarantees the client to have the exact same setup we did. However, this image is large and clunky, so in some this may be impractical. Setting up an appropriate containerized environment for testing is slow and tedious. To streamline this process, and ensure a proper testing environment that is supportive of our tests we created three installation files.

The first installation file installs Curl, Docker, and Minikube. Curl is a data transfer tool required for the installation of multiple tools we use. Once Minikube is installed, the current user needs to be able to access docker to be able to run the Minikube cluster. Our next script gives the user these permissions. Finally, the third set-up script installs Kubectl and Etcdctl, which are used for analysis of the cluster. Etcdctl is injected into the Minikube VM via Minikube ssh so that it has proper access and connection to etcd. Kubectl's configuration is also altered such that it will work with Minikube.

To run installation these three files must be run separately in order: *setup1.sh*, *setup2.sh*, then *setup3.sh*.

VII.5 General Secret Vulnerability Screening

The file secrets_tests.py contains a few general secret vulnerability testing scripts that probe for potential bad practices that could lead your secrest being insecure. These tests have been designed such that they can be as generalistic as possible, allowing them to be run on essentially any kubernetes system as long as kubectl is installed and properly configured.

The first test creates a new Kubernetes secret with plaintext. This tests to ensure that secrets are being encoded to base64 by default

The second test checks to ensure that secrets are not located within unsafe namespaces. As we do not know the configuration and namespaces within our client's Kubernetes system we simply included the default namespace. This can be easily expanded on to include other namespaces that are considered insecure.

The third test checks for leaky secrets. These are secrets that are stored in one namespace, yet accessible to another. This test failing indicates a serious issue, as containerization would be being broken.
To run these test run *secrets_tests.py*

VII.5 Etcd Secret Encryption Attack

This demonstration shows an attacker's ability to gain access to sensitive information within a cluster. Etcd is a key-value store used for Kubernetes cluster-state, configuration data, and metadata. It is critical to Kubernetes functionality and stores all Kubernetes secrets. By default, etcd is entirely unencrypted. This means if an attacker gains access to etcd, they gain access to a huge amount of sensitive data regardless of its namespace. This can include passwords, tokens, etc. To prevent this, proper encryption is essential. Our etcd encryption attack simulates an attack on etcd and attempting to access a secret. First our test creates a new secret with a known name and value. Next it attempts to access the location within etcd where the secret is stored. Finally, it traverses the data, searching for the secret's value and reports back if it found it. This can be used to test the effectiveness of various encryption configurations and ensure that they are properly applied.

Due to the necessity of using etcdctl, which is located within the Minikube vm, this tool is only intended to work within our environment. We also include a sample encryption configuration for Minikube, as well as a script to start it without encryption.
To start Minikube with sample encryption run *enable_encryption.py*
To start Minikube without encryption run *disable_encryption.py*
To run the test run *etcd_demo.py*

VIII. Product Delivery Status

Our work can ban found on the github repository at the following link:
https://github.com/WSUCptSCapstone-Fall2022Spring2023/nuwc-cybercontainers

Our clients will be able to access the final project there

IX. Conclusions and Future Work

IX.1. Limitations and Recommendations

The main limitation we as a group are experiencing at the moment is the lack of public information on the exploits we are researching, as it is in the best interest of Kubernetes developers to keep certain details about them a secret until they can be patched. This means reproducing them will be difficult, and some of our findings will undoubtedly be purely theoretical rather than tested and proven by the end of the project, as there will be cases where we can

make reasonable assumptions and prove an exploit possible, but may not be able to actually carry it out. As the project enters its final months it would be beneficial to adopt a kind of 'throw everything to the wall and see what sticks' approach, as going forward there will be much less documentation to work off of, and we will need a much larger knowledge base to support our findings.

IX.2. Future Work

Major tasks to complete in the future include:

- Look into cyber threats using the containerized software
    - Backdoor threats
    - Brute force threats
    - Man in the Middle threats
- Clean up the Github repo
- Develop robust threat analysis
- Expand repertoire of vulnerabilities we can demonstrate

X. Acknowledgements

We would like to thank the Naval Undersea Warfare Center, Benjamin and Sandip for the opportunity to work on this project and their guidance. We would also like to thank Professor Ananth Jillepali for his guidance and coaching during our work this year.

XI. Glossary

Containerized: Run in a discrete environment set up within an operating system specifically for that purpose and allocated only essential resources.
Cyber: Relating to or characteristic of the culture of computers, information technology, and virtual reality.
E2E: Kubernetes End-to-End Testing Framework
Emulator: Software that enables one computer system to behave like another computer system. An emulator typically enables the host system to run software or use peripheral devices designed for the guest system.
Docker: Is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.
Kubernetes: is an open-source container orchestration system for automating software deployment, scaling, and management.
NUWC: Naval Undersea Warfare Center

Secret: A Kubernetes object storing sensitive pieces of data such as usernames, passwords, tokens, and keys
Simulation: Production of a computer model of something, especially for the purpose of study
Virtual Machine (VM): The virtualization/emulation of a computer system.
WSU: Washington State University
OS: Operating System
Kubectl: A command line extension allowing for management of local Kubernetes clusters within a larger Kubernetes architecture.
Minikube: An extension that utilizes Kubectl and provides additional management functionality of local Kubernetes clusters.

XII. References

1. X. Merino Aguilera, C. Otero, M. Ridley and D. Elliott, "Managed Containers: A Framework for Resilient Containerized Mission Critical Systems," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018, pp. 946-949, doi: 10.1109/CLOUD.2018.00142.

2. D. J. Bodeau, C. D. McCollum, D. B. Fox, and M. C. M. V. Mclean, “Cyber threat modeling: Survey, assessment, and representative framework,” *DTIC*, 04-Jul-2018. [Online]. Available: https://apps.dtic.mil/sti/citations/AD1108051. [Accessed: 21-Sep-2022].

3. Vamsi Chemitiganti, “Learn about Kubernetes Concepts and Architecture,” *Platform9*, 26-Aug-2021. [Online]. Available: https://platform9.com/blog/kubernetes-enterprise-chapter-2-kubernetes-architecture-concepts/. [Accessed: 10-Oct-2022].

4. S.M. Kerner, “Colonial Pipeline hack explained: Everything you need to know”, https://www.techtarget.com/whatis/feature/Colonial-Pipeline-hack-explained-Everything-you-need-to-know.[Accessed: 9-Dec-2022]

5. J.R. Tietsort, “17 Types of Cyber Attacks Commonly Used By Hackers”, https://www.aura.com/learn/types-of-cyber-attacks#:~:text=From%20a%20Hack-,17%20Different%20Types%20of%20Cyber%20Attacks,phishing%2C%20whaling%2C%20etc.). [Accessed: 9-Dec-2022]

6. Google, Kubernetes, “Kubernetes Component”, https://kubernetes.io/docs/concepts/overview/components/. [Accessed: 9-Dec-2022]

7. B. Reselman, “What is the difference between kubectl and kubelet in Kubernetes?”, https://www.theserverside.com/blog/Coffee-Talk-Java-News-Stories-and-Opinions/compare-Kubernetes-kubectl-vs-kubelet-when-to-use. [Accessed: 9-Dec-2022]

8. Testim, “A Detailed Look at 4 End-to-End Testing Frameworks”, https://www.testim.io/blog/end-to-end-testing-frameworks/. [Accessed: 9-Dec-2022]

9. J. Haley, “Demystifying containers, Docker, and Kubernetes”, https://cloudblogs.microsoft.com/opensource/2019/07/15/how-to-get-started-containers-docker-kubernetes/. [Accessed: 9-Dec-2022]