GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Project

submitted in partial fulfillment of the
requirements for the course "Applied Computer Science"

Secure End-to-End Workflows using Containers in HPC

Simon Hernan Sarmiento Sabater

Institute of Computer Science

7. Oct 2021

Georg-August-Universität Göttingen
Institute of Computer Science

Goldschmidtstraße 7
37077 Göttingen
Germany

☎ +49 (551) 39-172000
FAX +49 (551) 39-14403
✉ office@informatik.uni-goettingen.de
www.informatik.uni-goettingen.de

First Supervisor: Hendrik Nolte
Second Supervisor: Dr. Julian Kunkel

I hereby declare that I have written this thesis independently without any help from others and without the use of documents or aids other than those stated. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Göttingen, 7. Oct 2021

Abstract

To make computing on HPC Clusters more accessible to users with very sensitive data some workflows have to be more secure than usual. In our scenario, among other things, an attacker has `root` *privileges on the machine we use to connect and submit jobs to the HPC Cluster, the frontend. Thus, compromising everything we upload to the Cluster in clear text. This project explores a potential solution, which uses LUKS containers, Singularity containers and Vault. LUKS data containers, to transport and store data securely, on the way to the cluster, as well as on the cluster, at rest. To transport the program's software environment, we use encrypted Singularity containers. Both of these containers above need their respective encryption keys to execute or read data. We have to provide keys on the cluster without compromising them. Vault, as a key management system, is explored, to store encryption keys, apart from, but accessible to the cluster, until the user's workload is scheduled and executed. We test the workflow on three virtual machines and subsequently analyse potential threats and improvements.*

Contents

1 Introduction **3**

2 Basics **7**

3 Current Setup **15**

4 Design **17**

5 Implementation **21**

6 Conclusion **23**

6.1 Vulnerabilities concerning our workflow 24

Bibliography **27**

Chapter 1

Introduction

Nowadays, vasts amounts of data is generated every day, from the tracking of consumer behavior to the generation of large scientific data sets. The desire to process/analyse these large data sets and gain insight calls for big computational capabilities. High Performance Computing (HPC) Clusters, otherwise known as Supercomputers, provide such capabilities. HPC Clusters have been around since the dawn of computers and have evolved over the years. At its heart, HPC Clusters are a group of *off the shelf* computers that communicate with one another to share computational workload and thus achieve *High Performance*.

These HPC Clusters are commonly shared with a whole spectrum of scientists and students (physicist, mathematicians, biologists, molecular biologist, aspiring doctors, ...) that want to process big computational workloads. Workloads can range from simulating a star collision to analysing healthcare records and anything in between. In order for a user to be able to use these resources, it is required to submit a job to a scheduler. For our site at the GWDG, the scheduler is called Slurm and the job is a bash script. These jobs are not completed sequentially but concurrently, which brings some issues noted further below. The scheduler will then allocate needed computational time on available nodes and execute the job. The results can then be viewed after the computation is complete.

With such a shared resource there come some problems, for example with sensitive patient data that is clearly identifiable to a real person. Here, the law and the good scientific practice state strict requirements on how these data sets need to be handled. These requirements usually exclude shared systems, where users without the necessary privileges to these data sets have access to them.

Additionally, HPC Clusters, being very powerful, are a popular target for attackers, that could turn these resources to their own profit, or simply steal other users data or computations.

So there is a clear motivation to ensure data is handled correctly and responsibly.

This thesis tries to develop a workflow, where users of a HPC Cluster can compute and process their data with certainty that the data stays confidential, throughout the entire workflow.

Fortunately there are tools and methods to protect data, some are named here and explained in more detail in the *Basics* section. We will shortly visit encryption and HPC security in a broader view, before looking at some complete workflows that sadly do not cover all of our expectations.

The most obvious way to protect data is encryption. We have a great variety of encryption methods that make communication and protection very hard, if not impossible, to circumvent, but only if done correctly. In this thesis we will use *Linux Unified Key Setup (LUKS)* and *RSA (Rivest–Shamir–Adleman)*, both described in the next section.

The most secure HPC Cluster (or System) is the one that is not connected to the internet, so called *air gapping*. There are institutions that do this, for example military organizations, but this makes it very difficult for users to compute on these resources. So, for our use case, it is not reasonable to isolate from the internet completely. Controlled access is *only* possible via VPN or by a special login computer. Of course user management and security is very important for the admission to the network, so, for example, strong passwords and removing unused accounts has to be enforced. More detailed security methods are described in [1]. It does an analysis of an existing HPC Cluster and what vulnerabilities they found.

Once on the network, users can only access resources and submit jobs via another computer, the *frontend*.

In our scenario the frontend is compromised to the highest level, there is a malicious `root`, an attacker that can do everything on the frontend. This results in any data we store on the frontend, the user's *$HOME* folder, is readable to the attacker, so protection worthy data has to be encrypted before we upload it.

Since there is an increasing demand for computational power in order to be able to process sensitive data, there are incentives to make computations secure on remote locations, like our HPC System. A problem with remote computations is that we, as users, don't know with whom we are sharing these resources. There could be hundreds of users computing at the same time and some users could be interested in what data we are computing, even the provider of such services could have an interest. If the remote computer or remote host do not take precautions it could be possible to peak at our data for example over the RAM, even while we are computing.

Let's look at some solutions to make computations on remote hosts more secure.

Related Work

A method to provide desired isolation from other processes or users are *enclaves*. These are specific CPUs with special capabilities to encrypt data before it is written into RAM. This is called *Trusted Execute Environments*, TEEs for short. [2] analyses the requirements of secure computing on HPC

Systems using TEEs and describes why there is still more work to be done in order to make secure computing on HPC Systems secure and performant.

We have the same problem but since purchasing new CPUs for the Cluster is not viable, we decided to solve the issue by providing an entire secure computer for a single user at a time, but only for users that need this enhanced security. This achieves isolation of processes but could mean, that if a workload does not fully utilize the resources of their secure computer, we are losing overall performance. The setup of this computer is explained in more detail in *Current Setup*.

For more data intensive computations one has to store the data on the remote host or HPC System, this again is a potential security hazard. How can a user be sure that his data is not compromised when storing it on the remote, untrusted system. [3] is a solution to this problem. Through the use of a distributed storage system called Ceph it provides isolated islands of security for each user, our intended solution, but there are some drawbacks, some also explained in [3]. The interface to address files is a S3 - REST API, which is not what we are looking for, we expect our interface to be a *POSIX* filesystem, the *normal* Linux interface. The most important drawbacks are the latency and strain on the network that we would have if we implemented this solution. At our site the resources are distributed over several locations, several kilometers apart, the resulting latency is too big for a computation to wait on data and still be efficient. Even ignoring latency, if many users have to access their files on different storage computers, our network hardware would be a bottleneck. There is another, bigger problem. Let us assume we are users and our data is already stored on the HPC System, how do we access the data? Obviously with the encryption/decryption key, but remember our frontend is compromised, so the attacker would be able to read our keys and decrypt the data for himself. To clarify the last problem; We have to provide a decryption key on the HPC System without it being compromised on the way there, specially through the frontend. This is tricky to solve and the main problem of this thesis.

The additional administrative work is another reason we are implementing another solution.

Another approach that does not fully satisfy our requirements is [4]. Here a database is run inside a Singularity container to isolate the database but still remain performant. Singularity containers are used in our workflow but in a slightly different way, it's described further below. The database used, *MongoDB*, has capabilities to restrict access to users by authentication and authorisation. Sadly it does not provide a *POSIX* filesystem, a requirement to make the computation easy for our users. And again, the problem of accessing our data securely is not solved, since again we would have to upload our keys to the frontend, to submit the job.

There are solutions that solve the problem of computing on untrusted, remote computers, but none so far fulfill all of our requirements. We want users to have their known filesystem interface, be able to upload and access their data in a secure and user-friendly way, and of course minimize administrative work.

To recap, the problems that have to be solved:

- **Data security in transit and at rest**
  1. Data has to travel from the client computer to the HPC system securely
  2. Once on the HPC system, data has to remain secure until the scheduled job can process or read it
  3. After the computation is done, data has to travel back to the client
- **Data security during computation**
  For the computation data has to be accessible to the process, we have to provide the keys to the HPC System securely. The process and data have to remain secure and isolated from other users or processes.

Chapter 2

Basics

Now that we have a basic idea of what the problem is, we still need tools to make the ideas work. This chapter explains some technologies hinted above and some new ones that make this project possible. Small examples should illustrate the functionality. We start with encryption tools, more specific with *Linux Unified Key Setup (LUKS)*, which is also used by other technologies under the hood.

Linux Unified Key Setup (LUKS)

LUKS is used to encrypt volumes on hard drives, so that only a person with the corresponding encryption key can access the volume. We use LUKS because it stores all necessary operational data in the partition header, making it easy to move these data containers around and mount them at other locations. A short example of how one would setup, a container to transport data, explanations are inline and at the end:

```
# name for our container
container_name="testcont"

# the final container we will get
container_file=${container_name}.img

# who we are when running the script
username=$(whoami)

# where we will be able to interact with the
# contents of the data container
mount_path=/mnt/secure_workflow

# size of our container
```

```
size=30

# find an available loopdevice
loop_dev=$(sudo losetup -f)

# write size amounts of random bytes onto the new container file
# we want to build
dd if=/dev/urandom of=$container_file bs=1M count=$size

# create an encryption key
tr -dc '0-9a-zA-Z' </dev/urandom | head -c 32 > keyfile

# assign a loop device to our data container
sudo losetup $loop_dev $container_file

# create volume encryption with -c cipher, -s size of the key,
# to be encrypted volume and the key
sudo cryptsetup -c aes-xts-plain -s 512 luksFormat $loop_dev $key_file

# open/ create a mapping for our now encrypted conatiner
sudo cryptsetup luksOpen $loop_dev $container_name --key-file $key_file

# make ext4 filesystem onto the mapping
sudo mkfs.ext4 /dev/mapper/$container_name

# mount filesystem onto our desired location
sudo mount -t ext4 /dev/mapper/$container_name $mount_path/$container_name

# change rights for us to put data into the container
sudo chown -R $username $mount_path/$container_name

# put data in the container
echo "Hello encrypted container!" > $mount_path/$container_name/hello.txt

# unmount our container
sudo umount $mount_path/$container_name

# reencrypt the container
sudo cryptsetup luksClose $container_name
```

A short note on *loop devices* and *losetup*. *loop devices* are not real devices but are used to access files as if they were block devices, e.g. hard drives, or a file with an entire file system. *losetup* simply manages and interacts with *loop devices*. *cryptsetup* is the utility to setup the volume encryption that we want. The fact that a user could potentially upload his whole project structure as it is onto the HPC System makes *LUKS* ideal for our use case. Encryption would happen on the user's computer; transport the data onto the HPC System, then, decryption with the same key would deliver the desired environment for the user to compute.

We know from above that the decryption on the HPC System tricky is, since our keys would have to be on the frontend. So this alone does not solve our problem.

Having presented the way we will transport data onto the HPC System, we still need to transport and execute our programs, to analyse the data for example. Before we look at a solution for that, we need to look at the *RSA (Rivest–Shamir–Adleman)* cryptosystem briefly.

RSA (Rivest–Shamir–Adleman)

This crypto-system is an asymmetric one, which means encryption is done with one key and decryption is done with another key. These functionalities make it possible to send data to a recipient without having to exchange encryption keys beforehand. Additionally, it can be adapted to work as a signature, this will not be used here though.

Since we will use this system to encrypt a *Singularity container* and afterwards run it on the HPC System, here a small example on how the keys would be generated:

```
# generate a key pair, of type (-t) rsa,
# of lenght (-b) 4096 bits, format (-m) pem,
# empty (-M) passphrase "" and store it the
# current working directory in rsa
ssh-keygen -t rsa -b 4096 -m pem -N "" -f ./rsa

# transform the public key (rsa.pub) into format (-m) pem
# and save it under rsa_pub.pem
ssh-keygen -f ./rsa.pub -e -m pem > rsa_pub.pem

# to have uniform naming we rename the rsa key to rsa_pri.pem
mv rsa rsa_pri.pem
```

The public key does not have to be secret, it's the way someone would encrypt a message for you, the holder of the private key, the only way to decrypt the message. We will use this functionality to encrypt our *singularity container* with the public key and execute it on the

HPC Cluster with the private key. Again we have the same problem as above, since again we need the key on the Cluster.

Containers, broadly speaking, are programs that isolate and run other specified programs inside. Through the use of *namespaces* and *cgroups*, the program inside the container uses the same host kernel and the performance is very similar.

The advantage to just running the program directly on the host system, is that you can package all your needed libraries and dependencies inside the container. Meaning, even if the host system does not provide your needed libraries, you can still have them inside your container. This is a very powerful tool since all users could package all their needed tools without the HPC System needing to hold a lot of software, thus decreasing vulnerabilities for the Cluster and increasing reproducibility and convenience for the users.

The most popular containerization application is *Docker*. *Docker* has found a lot of use cases, especially in microservices, like running a website or a database. For us, *Docker* is not an option, since the *Docker daemon* is always running and it is running with `root` privileges. Even so this workflow could be possible with docker, there are more convenient tools.

Singularity Containers *Singularity containers* have been developed with HPC in mind. They provide the same privileges inside the container as the user who started the container. These containers can also access resources of the host system, like file systems and graphic cards, fairly easy.

There is another very useful functionality of *singularity containers*, the ability to encrypt the whole container. This works by providing a rsa key pair, like described above and encrypting the container with the public key when you build it, then to execute it you provide the private key. The tool working under the hood here is the before mentioned LUKS. To use these containers, users would build and encrypt their own container locally, with all their needed tools to analyse or compute. Keep in mind that building a container requires *root* privileges or the use of *fakeroot*. Then the encrypted container is send to the cluster and executed with the respective private key.

Assuming we have generated the keys explained like above, meaning we have a `rsa_pub.pem` and `rsa_pri.pem` file, we now want to build a container. To build a container, one needs to specify a definition file or recipe like:

```
Bootstrap: docker
From: ubuntu:20.04

%runscript
    fortune | cowsay | lolcat
```

```
%post
    export DEBIAN_FRONTEND=noninteractive
    apt-get -y update
    apt-get -y install fortune lolcat cowsay

%environment
    export PATH=$PATH:/usr/games
    export LC_ALL=C
```

With the above file saved as lolcow.def we can execute:

```
sudo singularity build --pem-path=rsa_pub.pem lolcow.sif lolcow.def
```

This will deliver a `lolcow.sif` file and it can only be run like:

```
singularity run --pem-path=rsa_pri.pem lolcow.sif
#expect something like
 ________________________________________
/ Q: What's tiny and yellow and very,   \
| very, dangerous? A: A canary with the |
\ super-user password.                  /
 ----------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

We are now able to send our data without problems onto the frontend, since data and programs are encrypted. Now, to slowly address the biggest problem, how do we execute or access encrypted data in the HPC System if we do not trust the frontend.

We need a system apart, that only manages the keys that users need for their computation on the HPC System. So it has to be accessible from the user's computer at home, for users to upload their keys and on the other hand the HPC Cluster needs a connection to read the keys and execute the jobs.

We investigate *Vault* as a candidate to solve these problems.

Vault

Vault advertises itself as a *secrets management system*. A system where one can store any data that should only be read by the right person or program.

Through the use of policies *Vault* can restrict privileges in a very fine-grained way, for example we only want users to be able to write and read keys that are relevant to them, otherwise any user could be a potential attacker. A computational node on the HPC System should only be able to read the secrets/keys needed for the computation, and then not really be able to do anything once the computation is done.

The communication between the user or executing node and *Vault* has to be secure as well. *Vault* supports secure communication via TLS, making it secure to upload and read keys. The secrets at rest, so once the key is on *Vault*, it has to remain secure, so *Vault* encrypts everything before it is written to permanent storage.

Vault supports many authentication methods, here we will only use tokens to authenticate. The tokens for the users have to be provided by the system administrator. Users have to authenticate when uploading keys, and the HPC System or the computer currently processing the user's job has to authenticate to *Vault* to retrieve the keys. The latter is challenging, we will talk more about it in the *Design* section.

To transmit keys we use the *key-value* secret engine *Vault* provides. One can upload keys into a directory and read them if the permissions allows it. A short example:

```
# set Vault address example
export VAULT_ADDR='http://127.0.0.1:8200'
# set authentication token example
export VAULT_TOKEN=s.AnWnXCCrkBpMYBetFqgn3drf

# to upload a key
vault kv put secret/<user_name>/<identifier> <key>=<value>

# to read a key
vault kv get secret/<user_name>/<identifier>

# to read a key and get only the value
vault kv get -field=<name_of_value> secret/<user_name>/<identifier>
```

In our later described workflow we use some arbitrary username to distinguish users, the shasum of the encrypted container as identifier and key as name_of_value.

Lastly, an important feature used in this project is the *wrapping* and *unwrapping* of tokens. A token can be wrapped inside another token. This wrapping token hides the *important* token,

until it is unwrapped. Unwrapping can only be done by *Vault* and only once! This feature makes it easy to detect if a token has been compromised, initiating further investigation.

To solve some issues that emerged over the course of this project, we needed a way of filtering API requests made to *Vault* by IP-address. We decided on *nginx* to solve this problem.

Nginx

Nginx is a web server that can additional be used as reverse proxy or load balancer. For our specific use case, we needed to filter API requests to *Vault* by IP. Specifically filter user requests and HPC system node requests, so that only the HPC system has permission to make certain request, for us it was reading and unwrapping of tokens. *Nginx* provides the *ngx_http_geo_module* module, which does exactly this. It makes it possible to whitelist a set of IP-addresses for certain functionality.

These are the tools used in this project, and the presented workflows and examples resemble a user's workflow to compute securely on the HPC Cluster. Some are hidden from the user in scripts, like the encryption of data containers using LUKS, and some should be contained in the run script, like the *Vault* unwrapping.

These technologies, not all, are used in our current workflow as well. We will adopt the setup of the computational nodes from the current setup. Nonetheless, to motivate a possible change we look at the current workflow in more detail.

Chapter 3

Current Setup

We will go through an example workflow of the current setup and describe the parts as they come up.

The user, on his secure computer, will encrypt: data input-, data output- and singularity container as described above and upload these onto their $HOME storage on the HPC System.

Then the user will schedule a job that does nothing but reserve a secure node. A few notes on these secure nodes.

To achieve security on these nodes, `sshd` has been configured to lockout `root` login and only allow login with `ssh` keys of authorized users, these are public keys. These authorized keys have been configured at installation time of the secure node and communicated Out-of-Band, from user to administrator. So if the users keep their private keys safe, they are the only ones that can log in. To further mitigate vulnerabilities the dynamic loading of modules is disabled and only necessary modules, those to ensure encryption and necessary communication, are loaded. Lastly, because this also removes the possibility to configure or monitor the node as an administrator, all ports that are not relevant for the workflow are blocked. Only users can log in now. The secure node provides scripts to mount and umount data containers, so that no user has capabilities to use mount in another context. This excludes path-traversal-attacks as well.

Back to the workflow.

So after the job is submitted to Slurm the user has to wait, until the job is scheduled. This part is the one we want to change, and set it up in a way that it is handled by the HPC System.

When the job is scheduled on the node we, as the user, are the only person that can log into this secure node. We can upload our encryption keys for the data and singularity container with *scp* onto a temporary file system provided on the secure node, and schedule a job with Slurm to do the actual computation, the run script, which is also sent via scp directly onto the secure node.

We will keep the secure node setup, since it provides isolation from other users and a secure environment. This method has some downsides that are not solvable, but are good to be aware of. As mentioned in the *Introduction* we could potentially have unused resources. Additionally, we are blind to any problems on the node, we have no way of figuring out if the node is ok or has been failing for quite some time. We have to wait for a user to complain.

As already mentioned we want to change the user sided wait to upload the keys and execute the real computation.

Chapter 4

Design

Our target is a system to manage and hold encryption keys on behalf of the user. This would remove the need for the user to wait on a signal to upload the needed keys. The HPC System would communicate with the *new* key management system also removing the need for users to log onto the secure node.

First Iteration

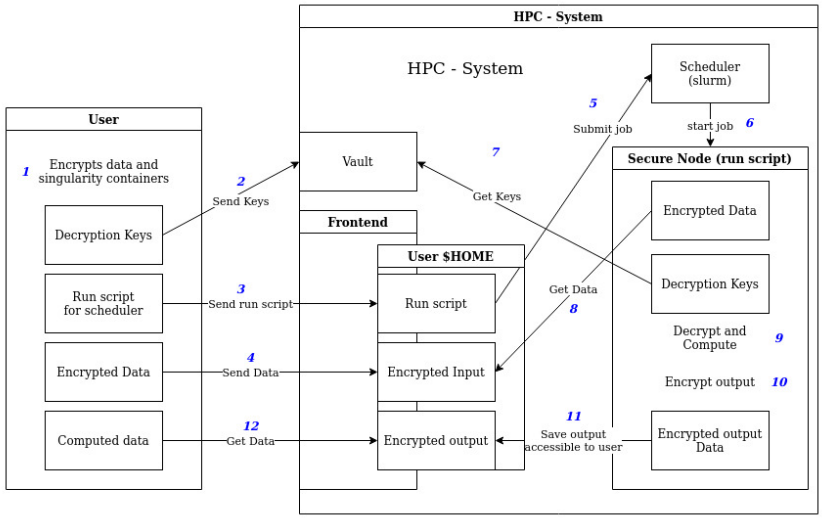

Figure 4.1: Initial design. The Numbers 1 through 12 are the individual workflow steps.

Explanation of the initial design is aided by picture 4.1. **We assume a secure user computer, otherwise everything is already compromised.**

The user would design his computation locally in a singularity container with two directories, one

with inputs and one with outputs, mounted and accessible to the container. Encrypting Singularity container, Input- and Output Data containers the transmission to the HPC frontend is secure. So far the workflow is the same as the existing one. The encryption keys to execute the computation and data access are submitted to Vault, with the hashsum of the encrypted container as identifier to retrieve the key. Lastly, a run script is written by the user, uploaded to the users $HOME and submitted to the scheduler (in this case *Slurm*).

Initially every node had a Vault authentication Token to fetch the keys needed, provided by an administrator.

The scheduler would look at the run script and start **only** the run script on the secure node. This script needs to fetch the needed data from user $HOME and the keys provided by the Vault instance, execute the singularity container, encrypt the output, save said output to user $HOME and exit. These operations are very similar to the ones described in *Basics* and the current workflow.

There are some problems with this initial workflow. Some pretty severe.

1. Any rogue user could extract the authentication Token from the secure node and from that point on everything is compromised. This would not be easily detectable and could be done from any external computer.
2. Any user could read out the secrets of other users.
3. A root user on the HPC Frontend. This poses a great problem. A root user on the Frontend could copy the run script and the data containers, run it as his own job, reencrypt the unencrypted data with their own keys and subsequently compromise the data.

Second Iteration

The first problem above could be solved by making the user provide a token to read the keys in their run script, so the secure node would no longer have a token that could potentially be compromised. Through the use of *wrapping* tokens, the user could send a generated token in the run script. This token can be unwrapped only once. Even if an attacker steals and unwraps the token it is detectable because the *correct* job would fail, since it can not provide a token with the right privileges. With a setup like this, an attacker could still unwrap the token from the external network, not telling us anything about who is doing the attacking. To regulate unwrapping and gain information on potential attackers *unwrapping* should only be possible from the internal network, more specific a secure node. While we are restricting access based on IP addresses, we can as well restrict read operations to only the secure nodes. We used Nginx to filter the IP addresses. This way, the attacker has to make the request for keys to *Vault* while on a secure node.

The obviously needed separation from user keys is provided by stricter *Vault* policies, resulting in *bubbles* of security for every user.

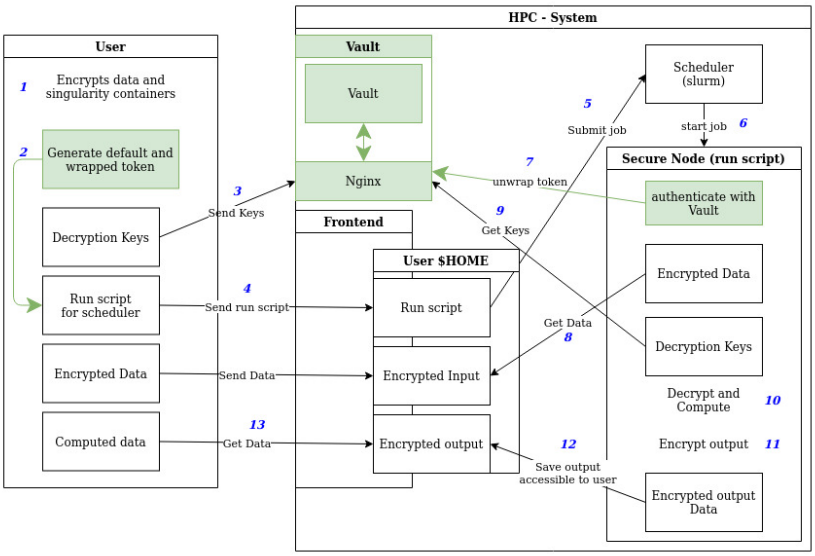

Figure 4.2: Second iteration design, new parts in green. The Numbers 1 through 13 are the individual workflow steps.

These tweaks result in the second iteration, 4.2. To upload the keys the workflow is the same, but to read them out from the node the user has to generate 2 tokens. A default token and a wrapped token with read permissions (Both tokens need to be in the run script). In the run script, the secure node authenticates with the default token to unwrap the token, then it authenticates with the unwrapped token to read the keys.

This solution is not fully secure, more on that in *Conclusion*.

Chapter 5

Implementation

Testing [1] of the proposed design is done on three virtual machines, one for each involved entity. The first (UN) is used by a user, this would be the user's home computer or laptop. Second the key management system and the unsafe shared storage (VFN), this one mixes both the frontend and the Vault-nginx functionalities. In a production environment these should be separated! Third is the compute node (SN), this one only computes when it is given something to compute.

The VFN node has a *Vault* container running in which for every user two policies are build. One to read and unwrap, the node token and another to generate these node tokens and write keys to *Vault*. The *root* token, a token that can do everything is disabled, and can only be enabled if the unseal key, is provided. In a production environment this would be a combination of 3 keys that are distributed to different people.

With *nginx* setup as reverse proxy and IP Filter using *ngx_http_geo_module* module, we can whitelist all necessary IP addresses. The whitelist for this test consists only of the SN IP address.

After the setup of *Vault* is complete, the user is provided with his token.

The user encrypts his data, uploads it to unsafe shared storage on the VFN. Ssh access between UN and VFN is needed to transfer the file via scp. The keys are uploaded to *Vault*, after setting authentication and location environment variables *VAULT_TOKEN* and *VAULT_ADDR*.

As Slurm is not used in the test environment, we had to start the job manually via ssh, so additionally ssh access from UN to SN has to be installed.

Once the run job starts, it executes as described in *Design*. It authenticates with *Vault*, fetches the encrypted data from unsafe storage, fetches keys from *Vault* and executes the singularity container, encrypts the output and safes it to unsafe storage.

[1] The whole implementation is available on `https://gitlab.gwdg.de/s.sarmientosabater/secure-workflow-testenv`.

Chapter 6

Conclusion

This thesis develops a secure workflow to compute sensitive data on a HPC System, naturally we have to ask *how secure is it?* and *is there a better way?*

For every tool we use, we inherit the vulnerabilities as well.

Vault specifies many threat vectors in [5]. To start, an attacker with privileges to the storage backend could completely disable the service, by deleting everything, for example. This would not compromise keys or data, but it would make *Vault* inoperable.

Since *Vault* is open to users to submit requests, it is also open for attackers to try a distributed denial of service attack. This would only work from within the network, reducing the potential strain and volume of requests substantially.

Even though we are filtering IP addresses, an attacker could spoof their IP and make it seem to *Vault* and *Nginx* as if a secure node is making a request to unwrap a token or read a secret. This is not a problem, *Vault* would respond with the unwrapped token, but the unwrapped token would be sent to the secure node, not reachable to the attacker.

More dangerous is an attacker with access to RAM on the Vault system, this could compromise data by inspecting it before encryption is possible. To access the *Vault* instance an attacker would have to gain access to the administrative network, then access the computer on which *Vault* is running and then gain `root` access. This threat could in theory be only exploitable by administrators.

Singularity mounts unencrypted volumes in read-only, thus protecting the software environment at every moment in time. For Singularity, there is a known vulnerability concerning non encrypted swap. If the Singularity container writes to unencrypted swap, this can be inspected by an attacker. A simple solution is to disable swap on the image before it is even deployed, like in our current setup. Additionally, our current setup, isolates an entire node, there is no one who could inspect it. Further, the definition or recipe file in the container is not encrypted, so if sensitive information is located in this file it has to be deleted from the container file.

Attack-vector	**Solution**	**Current setup**
Intercepting packages	Data is encrypted, and TLS is used	Data is encrypted
exploitation of user's keyagent	no keyagent involved	Through the use of the `-c` option
privilege escalation on the frontend	Data is encrypted, **the run script is in clear text compromising the wrapping token**	Data is encrypted
privilege escalation on the secure node	not possible, since no ssh is needed	Data is encrypted until the legitimate user uploads keys, when he is the single tenant
privilege escalation on frontend, impersonation and submission of malicious job	after stealing a wrapping token, this **compromises data** and is detectable	single tenant reservation of secure node has to be longer than the data is accessible
IP spoofing to impersonate a secure node and requesting unwrapping of stolen token	not viable since response is send to the secure node	Vault is not used
accidental misuse	incorrect job submission	incorrect job submission, compromising keys by uploading not using provided script
Social engineering	token security has to be clear to the user	private key security has to be clear to the user

Table 6.1: Threat vectors, compared to current setup

6.1 Vulnerabilities concerning our workflow

If encryption is done correctly, data containers can be transferred onto the HPC System securely. The key transmission has to be secure, so TLS should be used in a production environment. This way any outside attackers can not read data even if they intercept the packages.

A possible attack scenario could be as follows. We have uploaded the data as a user to our $HOME and have scheduled a job on a secure node, the job has not been executed yet. Assuming somehow the attacker has gained `root` privileges on the frontend. He would have to copy and modify the legitimate run script to extract our encryption keys, for example coping them to a $HOME in clear text. He would then impersonate us and submit his modified run script to execute on a secure node. The attacker's job has to be scheduled before our legitimate job, but since the attacker has no interest in doing computational work, this should not be a problem. During the execution of the job, the keys can be intercepted. The attacker is now in control of the encrypted data as well as the encryption keys.

The described attack would make use of the unwrapping mechanism *Vault* provides and since the request is originated on a secure node it will, as intended, unwrap the token. This will result in the legitimate job not being able to request unwrapping, since this operation is only permitted once, the user's legitimate job will fail. Therefore, a potential data leak can be detected.

The underlying problem here is our methods to ensure that the legitimate user is submitting a job and not an impersonator, is not robust enough to withstand an attacker with `root` privileges on the frontend, as described above. This calls for authentication of the run script, to verify the user did in fact submit this specific run script.

For this we could identify two possible solutions. First is to let the users sign each run script locally, with their private key, and then let Slurm verify it with the user's public key. Second, a two factor authentication (2FA) with the user to verify the authenticity of the submitted run script. Both could still compromise our data if we find ourselves with a rogue or hijacked user, that has acquired a wrapped token and is determined to reveal his malicious intentions in exchange of compromising data. In case of an attack we could pin point the malicious user by reviewing the SlurmDB and Vault/Nginx logs.

This could be mitigated by having encrypted run scripts, that only Slurm can read. By having users encrypt the run script with Slurms public key, Slurm is the only actor that could run the scripts.

All the proposed solutions above depend on Slurm not being compromised. Additionally, if a 2FA authentication is implemented the SlurmDB has to be secured, as it is treated as a trustworthy entity. This is a reasonable assumption, since no users can access the administrative network and the node.

The project went a step in a more secure direction, by reducing the number of threats, table 6.1, and making misuse of the workflow and accidental data leakage more improbable. Still, if a user fails to follow the workflow and forgets to reserve single tenancy on a secure node, the job will execute on a vulnerable machine.

There is no longer the need for a user to upload their public key and log into the secure node. Potentially forgetting to set the `-c` flag when configuring their *key-agent*, which could lead to a user's key being hijacked, is also mitigated. All key actions are handled by *Vault* and the HPC System.

All in all, this workflow does not provide the desired security. It does however detect a malicious access to encryption keys, but at that point data has already been leaked. The analysis shows that in order to compute securely on the HPC System one needs to encrypt even the run scripts. A way to achieve this functionality could be found in [6] an is explored as future work.

Bibliography

[1] R. Bulusu, "Addressing security aspects for hpc infrastructure," *2018 International Conference on Information and Computer Technologies*, 2018.

[2] A. Akram, "Trusted execution for high-performance computing," *Eurosys Doctoral Workshop 2021*, 2021.

[3] A. Smith, "Exploring untrusted distributed storage for high performance computing," *Practice & Experience in Advanced Research Computing*, 2019.

[4] A. M. Dissanayaka, "A review of mongodb and singularity container security in regards to hipaa regulations," *Data-center Automation, Analytics, and Control*, 2017.

[5] HashiCorp, "Vault security model," *https://www.vaultproject.io/docs/internals/security*, 2021.

[6] Slurm, "Job submit plugin api," *https://slurm.schedmd.com/job_submit_plugins.html*, 2021.