



Abertay University

SCHOOL OF DESIGN AND INFORMATICS

Designing and Evaluating Solutions for Securing Trusted Research Environments

Sam Heney

supervised by

Dr. Natalie Coull

22nd January 2022

Acknowledgements

Firstly, I would like to thank Natalie Coull, my supervisor, for her continued support, encouragement, and care throughout this project. I chose this project because of how interesting Natalie made it sound, and while it was initially daunting, she has really made the process of creating this dissertation an incredible one.

I would also like to thank my friends in the cozy zone, as we spent many, many hours working together and procrastinating together as we all marched on towards the completion of our respective dissertations/work. I don't think I would have been able to maintain my motivation without that sense of camaraderie.

Finally, I would like to thank my incredible partner Izzy for all her love and support. This project has at times been stressful and intimidating, but she was always there to provide me the comfort and support I needed whenever I needed it.

This project would not have been possible without any of these people, and I have endless respect, love, and appreciation for them all.

Abstract

This project examines the security of Trusted Research Environments (TRE), which are used to give researchers access to patient medical data in a secure and controlled environment. Firstly, issues with current TRE infrastructure are researched, the risks that these may bring are identified, and potential solutions are discussed. Then, during the two part methodology, a more detailed look at how these solutions may be implemented is explored, and three proof-of-concept (POC) solutions are implemented and deployed. All three of these POCs were successfully implemented, though the third could have been better integrated with the other two given more time. The effectiveness of these solutions for mitigating the identified risks is then discussed, as well as how they might be integrated into existing TRE infrastructure.

Contents

1	Introduction	1
1.1	Background	1
1.1.1	Patient Data and Privacy	1
1.1.2	Trusted Research Environments	2
1.2	Aim	4
1.3	Research Questions	4
1.4	Objectives	4
2	Literature Review	5
2.1	Patient Data and TREs	5
2.2	Software Defined Infrastructure	7
2.3	Log Analysis	8
2.4	Summary	9
3	Methodology: Design	10
3.1	Identifying Risks	10
3.1.1	Code Ingress	10
3.1.2	Containerisation	11
3.1.3	Egress	11
3.2	Code Ingress	12
3.2.1	Activity Monitoring	13

3.2.2	Git Diode	13
3.2.3	Log Aggregation	14
3.3	Containerisation	15
3.3.1	Private Docker Registry	15
3.4	Data Egress	16
3.4.1	File Formats	16
3.4.2	Malicious ML Model Detection	17
3.4.3	Logging Code Execution	17
4	Methodology: Implementation	19
4.1	Creating a Testing Environment	19
4.1.1	Hypervisor	19
4.1.2	Research Environments	20
4.1.3	Networking and Firewalls	21
4.2	Git Diode	21
4.2.1	Ingress Share	21
4.2.2	Scanning Script	22
4.2.3	GitLab Deployment and CI Pipeline	23
4.3	Activity Monitoring and Log Aggregation	25
4.3.1	Logging Process Execution	25
4.3.2	Deploying ELK Stack	27
4.3.3	Sending Logs	29
4.3.4	Visualisation	30
4.4	Automated Egress Checks	33

4.4.1	Egress Share	33
4.4.2	Automatic Scanning	33
4.4.3	Verifying File Type	35
5	Results	38
5.1	Network Diagram	38
5.2	Topology Overview	39
5.2.1	Research Environments	39
5.2.2	GitLab Ingress	39
5.2.3	ElasticSearch Cluster	40
5.2.4	Admin Interface	40
6	Discussion	41
6.1	Git Diode	41
6.2	Activity Monitoring and Log Aggregation	42
6.3	Automated Egress Checks	44
6.4	Evaluation of Success	45
6.4.1	Aim	45
6.4.2	Research Question 1	46
6.4.3	Research Question 2	46
6.4.4	Research Question 3	46
7	Conclusions	47
7.1	Future Work	48
A	Appendices	51

A.1	Git Diode Ingress Script	51
A.2	Kibana Dashboard	52
A.3	Egress Scripts	53
A.3.1	scan bash script	53
A.3.2	type_check Python script	53

List of Figures

1.1	Typical TRE configuration	3
3.1	Manual code ingress	12
3.2	Git diode design	14
4.1	Proxmox user interface	20
4.2	Fresh Windows install in VM	20
4.3	Three template copies deployed	21
4.4	Some basic firewall rules	21
4.5	Ingress share configured	22
4.6	Windows share mounted	22
4.7	Repository for a research project	24
4.8	Runner shown in the web interface and in the runner list on the server	24
4.9	New file being added to the project, pushed, then ingressed	25
4.10	Process tracking enabled	26
4.11	Sample of process logs	26
4.12	Sample of the detail of each logged event	27
4.13	es-1 node configuration	28
4.14	Elasticsearch nodes deployed	28
4.15	Count of log sources	29
4.16	Relevant log sources after searching	29

4.17	Winlogbeat config	30
4.18	Winlogbeat index	30
4.19	Kibana Discovery page	31
4.20	Kibana Python executions lens	31
4.21	Process list	32
4.22	Python starting a Firefox process	33
4.23	Sample du output	34
4.24	Scan script working	34
4.25	Sample of the type list json	35
4.26	Scanning changed files	37
4.27	Cron job	37
5.1	Diagram of final infrastructure	38
5.2	Research Environments	39
5.3	GitLab Ingress	39
5.4	ElasticSearch Cluster	40
5.5	Admin Interface	40
A.1	Kibana dashboard with three lenses	52
A.2	Kibana dashboard filtering for child processes of powershell	52

Chapter 1

Introduction

1.1 Background

1.1.1 Patient Data and Privacy

Patient data is medical information and history of specific patients. This may include which medications they are taking, conditions they are suffering from, lifestyle choices that may be influencing their conditions, and how all of these things change over time.

All of this specific patient data on its own isn't particularly useful other than for diagnosis for that specific person and analysing their health. However, when data from several different patients is pooled together and examined, it may allow for the observation of trends or correlations between medication, conditions, lifestyle choices and other potentially unknown factors.

For example, a study of patient data was used to disprove the notion that the MMR vaccine caused autism (Smeeth et al. 2004). This mitigated the drop in vaccine coverage seen at the time due to the misinformation spreading, surely leading to many lives being saved.

In another case, a Scottish study demonstrated a correlation between less ill health of babies born after a ban on smoking in enclosed spaces (Mackay et al. 2012). This allowed the government to further justify the ban to the public, and know that they made the correct decision to enact the ban.

While available patient data is demonstrably beneficial to society, it also comes with privacy concerns. If anyone claiming to be a researcher has access to any patient data, then anyone can gain access to any person's medical records, an obvious breach of privacy.

There are several ways to mitigate against the privacy issues surrounding patient data. First, there's the process of anonymising the data, that is to take the initial data

set of patients and extract only the data relevant to the research, while ensuring that nothing provided to the researchers can be used to identify patients.

For example, if a researcher asked for a dataset of patients that live at a specific postcode, have three missing fingers and a rare autoimmune disease, it's highly likely that they are targeting a specific person and are just attempting to access their medical records.

Conversely, if a researcher asks for all those taking a specific heart medication and those who have recently had a stroke, that can be anonymised to only those two factors. That dataset is also understandably useful as it might provide insight into the correlation between those factors.

Unfortunately, anonymisation can never be perfect since the identifying characteristics of a data set are what is actually useful to researchers. Even with anonymisation controls present, it would still have grave implications on patient privacy if anyone could access these data sets, especially since it's often possible to reidentify patients by combining multiple data sets (Lea et al. 2016).

This is why Trusted Research Environments are necessary. It's important to have a controlled and heavily moderated environment where research projects can get access to necessary data in a secure and trusted way. Ideally, these environments will minimise the possibility of patient reidentification while simultaneously enabling researchers to do any research that they might need to do.

1.1.2 Trusted Research Environments

There are many different approaches to creating these research environments, but they all rely on a similar configuration. Typically there will be several different research environments, all of serve different projects. Those projects will have access to a limited data set relevant to their research project, and no access to any other data.

These research environments are virtualised and ideally should be isolated from one another. The virtualisation host will have access to the full data store and some access controls that determine who can access which environment. The researchers get access to the environment remotely. A diagram of this set up can be seen in figure 1.1.

This represents a simplified configuration, and in reality these systems are more complicated than presented here, but this illustrates the fundamental design pattern

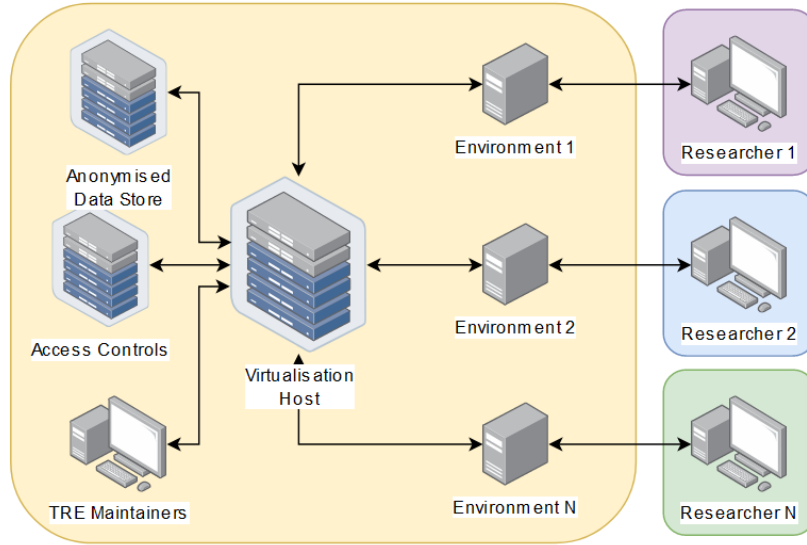


Figure 1.1: Typical TRE configuration

of these systems.

These research environments need to be carefully balanced between providing sufficient data processing functionality and being sufficiently restricted to minimise the possibility of malicious activity taking place.

For example, in the case of a system like figure 1.1 a malicious actor may wish to break out of the virtual environment using virtual machine (VM) escape techniques. This would potentially give them access to the other research environments as well as direct access to the anonymised data store, allowing them to access data they are unauthorised to access. Depending on the configuration of the infrastructure, it may also be possible for users to perform denial of service attacks by using up all available system resources within their environment.

Beyond attacks on the infrastructure, there are other potential issues to consider. If a user finds a way to bring another patient data set into the environment and combine it with the one provided, there is a possibility that aggregating the data will enable de-anonymisation. For example, if one data set is around a symptom and a medication, and another is about a symptom and a geographical location, these could be aggregated to find people specific to all three.

For this reason, it's extremely important to be careful about what can be ingressed into the environment. This project will go into detail about several other ways in which patient privacy could be compromised, but essentially the approach to securing TREs needs to be modular. There will not be one or two security solutions that solve

all of the problems, but each problem presented needs to be examined and potential solutions explored.

1.2 Aim

The aim of this project is to identify some missing security features of existing Trusted Research Environments, discuss potential solutions, create proof-of-concept implementations and evaluate their potential.

1.3 Research Questions

1. What are current TRE implementations lacking?
2. Of those lacking features, how can these be implemented securely?
3. How can existing technologies help improve security of TRE infrastructure?

1.4 Objectives

- Study existing infrastructure to understand the current landscape
- Develop an understanding of the threats to TREs, and how they might be mitigated
- Present the areas that current TREs are lacking in, specifically features that might be difficult to implement securely
- Design and develop some tooling or demonstrations of technologies that would enable the mitigation of some specific risks
- Evaluate the effectiveness of the proposed solutions, and how they might be integrated into existing infrastructure

Chapter 2

Literature Review

This chapter goes over the research that was performed at the early stages of this project. Firstly, a background of securing patient data and existing TRE infrastructure is examined. Then, various aspects of securing real infrastructure in a more general sense are researched, along with technologies that can be used to improve the security of TREs.

2.1 Patient Data and TREs

Patient data is extremely important for medical research. Using data science techniques on patient data allows for insights into medical science that wouldn't be possible with just using normal medical trials. Especially with modern techniques like machine learning, the more high quality patient data available to researchers there is, the more potential insights researchers will be able to make.

It can be difficult for a multitude of reasons to get consent from patients to do research on their data, so there needs to be an alternative. Anonymisation is part of the solution, which is where the data is stripped of all personally identifiable information. Unfortunately, this is an incredibly complex problem to solve and it usually can't be perfect (El Emam, Rodgers and Malin 2015).

In 2014 the EU advisory body Article 29 Data Protection Working Party released an opinion critically evaluating several data anonymisation techniques (Article 29 Data Protection Working Party 2014). In the opinion, they set out how important anonymisation is to the privacy of patients and that it's important to find usable solutions to anonymise data so that researchers can have access without issues.

However a critical appraisal of this opinion points out that while the group make a few valuable points, it is far too theoretical in its proposed legislative ideas to be practically useful (El Emam and Alvarez 2015). The opinion piece states that the only acceptable risk of re-identification is zero, but El Emam points out that in practice

this is nearly impossible. If the only allowed risk is zero, this means that the quality of most data sets reduces dramatically, and it becomes significantly more difficult to perform meaningful research with such minimal data sets.

As both parties point out, it is therefore important to not only consider the anonymisation of the data, but to control who has access to it. This is the problem that Trusted Research Environments, also known as Data Safe Havens, intend to solve. With authorised bodies overseeing research projects and providing them limited access to only the required data sets, the risk of privacy violations should be sufficiently minimal.

Many different Trusted Research Environment designs have been implemented, and they are predominantly deployed with traditional network design patterns. These are servers running on local premises, managed by in-house system administrators.

One of these for example is the SAIL databank, a national data safe haven created in 2007 (Jones et al. 2019). Even though it's quite old now it has continually been adapted and upgraded over time to adjust to new emerging technologies and to adapt to new legislation.

It operates on a privacy-by-design model where the safety and anonymity of the data is prioritised over all else. The control measures used to ensure this are built in to the system at a fundamental level, rather than "bolted on" after implementation.

This solution is built entirely using on premises servers, and they have a dedicated building for it. The servers are used for other data science initiatives, but the paper describes how the building has plenty of physical safeguards and other access controls that should keep it physically safe.

A significant amount of work has gone into updating the storage mechanisms and infrastructure to account for legislative changes, such as GDPR. This will inherently be an issue with older solutions as the legislation develops over time, but they have now implemented a framework for responding to legal changes, though if another change as significant as GDPR happened it would still be significant work.

Unfortunately, similar to most TREs, there are still some key missing features. Ingress and Egress are both extremely limited, and managed entirely with manual processes. Also, unlike some other solutions like AWS's Service Workbench, the tools available within the environments are limited to just what are provided by default.

There is also no mention in the study about any adaptations they've made to

more modern data processing techniques like Machine Learning. These present an interesting problem in that the valuable data to be extracted might be a binary blob, which cannot be easily manually parsed for maliciousness.

2.2 Software Defined Infrastructure

One important aspect of Trusted Research Environments is the ability to quickly deploy new environments for different projects, and having those environments be immediately ready for use. A commonly used solution for this is to make use of Software Defined Infrastructure, also known as Infrastructure as Code, which is to write code that can be parsed by a virtualisation environment which will then deploy the environment as described.

A paper discussing a similar scenario to TREs, *Unleashing Full Potential of Ansible Framework: University Labs Administration* (Masek et al. 2018), describes using Ansible for describing and deploying the infrastructure of University research laboratory hosts. Ansible is a tool commonly used for provisioning environments in a programmatic way, and can easily be scaled to provisioning large numbers of environments.

In this case, the authors have developed a system where they make use of Software Defined Infrastructure to manage a "large number" of computer classrooms. They created a system that enables them to deploy several different operating systems (Windows 10, Ubuntu 16.04, Debian 8) and install various programs to those hosts all completely remotely and at a large scale.

Their findings demonstrate how a similar infrastructure could be created for Trusted Research Environments. A researcher may need different platforms or tools depending on the data they are handling or what they are comfortable with, so it would be beneficial to give them access to those platforms and tools on demand.

A real example of this, but less generalised, is AWS's Service Workbench (Alexa 2020). In this case the AWS feature of CloudFormation templates are used to describe virtual research environments, which can then be deployed on demand.

However, compared to the previous paper's solution, this relies on Amazon's proprietary tools and infrastructure, which will come with the cost of paying Amazon for maintenance and support. In the case of Ansible, there is plenty of online documentation and support for free, as well as the freedom to deploy on any cloud platform or

local infrastructure.

2.3 Log Analysis

There are many options available when it comes to monitoring endpoint activity, including a vast number of commercial solutions. These function by taking logs from each endpoint, aggregating them into one place, and looking for events or patterns that might be considered anomalous.

Of the two main major log analytics platforms, splunk (closed source) and the ELK stack (open source), ELK has overtaken splunk in google trends as of 2013 (Google 2021). This indicates that the ELK stack, a free and open source product, is more widely used.

The paper "Performance of ELK stack and commercial system in security log analysis" (Son and Kwon 2017) addresses the question of splunk against ELK more directly. They note that the pricing of splunk is quite high, since it charges by the amount of traffic at a rate of thousands of dollars per gigabyte. This alone may make splunk unfeasible for many smaller businesses, or publicly funded businesses like Trusted Research Environments.

The researchers ran an experiment of ingesting 100 million entry to 1 billion entry log files, then testing the performance of each technology when querying for specific conditions. Interestingly, they found that ELK outperformed splunk in this metric. At the very least, this demonstrates that ELK is just as capable as splunk from a performance standpoint.

They also examined the visualisation capabilities of ELK, or rather Kibana which is the visualisation component of the ELK stack. They found that Kibana was completely sufficient for complex analysis and aggregation of data, perfect for a smaller business since they won't have to rely on finding other tools to integrate with the system, unlike with splunk.

Another paper, Automated Threat Hunting Using ELK Stack – A Case Study (Shibani and E 2019), examines how the ELK stack might be used to monitor threats in Windows endpoints. They conducted an experiment where three different threats were created on a network, to test if the ELK setup would catch them.

For the infrastructure, they deployed the entire ELK stack on one host, as well as

deploying a Windows Server 2016 and a Windows 10 client. The malicious actions were conducted on the client.

In order to pick up more information than the default Windows logs provide, they make use of the tool Sysmon. This is a third party tool distributed by Microsoft that allows for the logging of key extra information such as file edit time changes and process creation/termination.

Making use of Sysmon to generate logging and sending back those logs to the ELK stack with Winlogbeat, a tool to send logs, resulted in comprehensive coverage of activity. The researchers found that this enabled them to track and detect all three threats they created.

These two papers indicate that using an ELK stack is an ideal solution for log analysis, and that using Sysmon can enhance the monitoring of Windows environments like the ones in TREs.

2.4 Summary

TREs are fairly new and underdeveloped, and there is much more work to be done in many aspects of them. However, the problems they're trying to solve of patient privacy while enabling researchers are extremely important, and there is plenty of work being done.

The field is very fast moving both in terms of technology and regulation, so it's difficult to pin down specific issues to address. However, code ingress, access to technologies within the environment and data egress seem to be three very important problems to be solved.

Shifting TRE infrastructure to a more dynamic Infrastructure-as-code design pattern may be beneficial, and these ideas are already being picked up by Amazon with their Service Workbench solution. It's important for on-prem environments to keep up with these changes, and there are many solutions available for this, but deploying them will be tasking.

Finally, it seems that a key aspect of securing environments is to maintain a sufficient audit trail of user activity. This does not seem to be present in existing infrastructure. The ELK stack seems to be the ideal solution for this, since it's both widely adopted already and completely free to use.

Methodology: Design

This chapter has two key parts. Firstly, the risks of several key aspects of TREs are identified and categorised with a risk matrix. Then, various solutions and mitigations for each of these risks are theorised about, ranging from practical solutions to policy changes. The ideas here are informed both by findings from the literature review and conversations that were had with real TRE system administrators.

3.1 Identifying Risks

Each identified risk will be given a score based on the following risk matrix. The value for feasibility will be given first, then severity. These will be used to calculate the final risk score.

		SEVERITY				
		1	2	3	4	5
FEASIBILITY	1	1	2	3	4	5
	2	2	4	6	8	10
	3	3	6	9	12	15
	4	4	8	12	16	20
	5	5	10	15	20	25

Table 3.1: Risk Matrix

3.1.1 Code Ingress

- Code may be harmful/malicious, both intentionally and unintentionally:
 - VM Could be compromised allowing for persistent access outwith normal channels. ($F = 1, S = 5 \therefore R = 5$)

- Code could be used to perform a VM escape, allowing access to the host. This is the worst possible scenario. ($F = 1, S = 5 \therefore R = 5$)
- Data might be hidden in code files:
 - Arbitrary data ingress could allow for combining datasets to re-identify patient data, compromising privacy of patients. ($F = 3, S = 5 \therefore R = 15$)
 - Malicious binaries/binary data could be encoded in base64 or similar and ingressed ($F = 2, S = 5 \therefore R = 10$)

3.1.2 Containerisation

- Lack of isolation allows container to have access to several key resources:
 - Container shares kernel with host, potentially allowing for breakouts ($F = 1, S = 5 \therefore R = 5$)
 - Access to kernel also means if the container causes a kernel panic, the host is taken down too ($F = 2, S = 3 \therefore R = 6$)
- Poisoned Images:
 - Often applications will be pulled from internet in a pre-packaged image. If pulled from an unreliable source, it might be malicious ($F = 1, S = 4 \therefore R = 4$)
- Denial of Service:
 - Containers, by default, have unrestricted access to system resources. A malicious container could use up all the resources and slow the machine to a halt ($F = 3, S = 3 \therefore R = 9$)

3.1.3 Egress

- Data may be hidden in files using steganography:
 - Complex file types like pdf, docx, xlsx can be used to hide data ($F = 3, S = 5 \therefore R = 15$)
 - Steganography also allows data to be hidden in images and other binary data ($F = 3, S = 5 \therefore R = 15$)

- Machine learning models can also be used as a vector for hidden data ($F = 2, S = 5 \therefore R = 10$)
- Data may be obscured from detection with obfuscation:
 - Many different methods of encoding like base64, xor, hex ($F = 4, S = 3 \therefore R = 12$)
 - Encryption can also be used to hide data ($F = 3, S = 4 \therefore R = 12$)

3.2 Code Ingress

Code ingress is the process of allowing the user of an environment to take code they have written in a local environment and import it into the research environment that has access to the data they need to process. This allows for developing data processing solutions in an environment comfortable to the researcher.

After speaking with TRE maintainers, it's been determined that the ability to bring previously written code into the environment is an important feature, to the point that when it's not allowed the researchers will attempt to find workarounds. This will inevitably lead to insecure code being ingressed, as well as allowing for other risks like arbitrary data ingress.

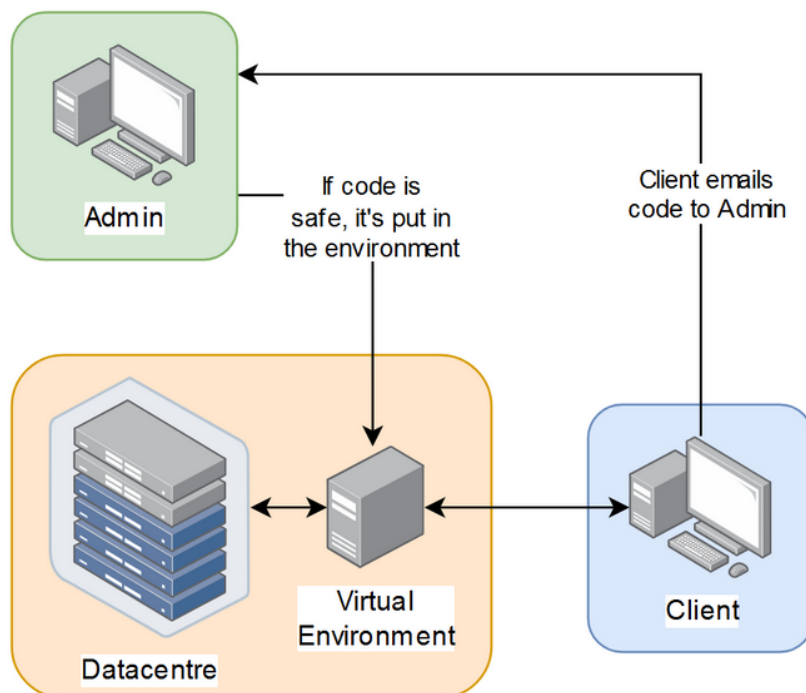


Figure 3.1: Manual code ingress

As shown in figure 3.1, code ingress is possible via manual review, where the code is sent to a team to check the code for potential risks or malicious behaviour before allowing it into the environment. This is not a scalable solution and even on a small scale could lead to security issues if the reviewing team is given too much to review and not enough time to be thorough.

Therefore, in order to be secure these environments need to provide a means of code ingress which is controlled, monitored and ideally automated.

3.2.1 Activity Monitoring

One finding of speaking to maintainers, is that users would sometimes make use of the paste functionality in the environment to copy and paste code from their local computer into the remote environment.

As long as the user can type in the environment, they could set up a local macro to type out whatever they have on their local clipboard. This would also bypass any code ingress protection methods, but is difficult to detect without sufficient monitoring in place.

One potential mitigation is to limit the areas that the researcher can edit files and monitor that restricted environment for changes. If a large amount of text is written to a file in a short amount of time, this indicates the use of a macro to type out code automatically.

3.2.2 Git Diode

The best way to mitigate researchers finding workarounds for code ingress systems is to make the in place system as accessible as possible. As mentioned previously, the system that is most prevalent currently is to email the code to a team for manual review. This is slow, arduous and error-prone, and very likely to motivate the researcher to look for a workaround.

One potential solution to automate code ingress is to make use of a git diode, that is a git server in between the client and the environment, that when code is pushed to can automatically scan and verify code before deploying it to the research environment. A diagram of this is shown in figure 3.2. This would create a simple interface for code ingress (git) which the researcher can use to quickly deploy code into the environment

potentially without any need for human verification.

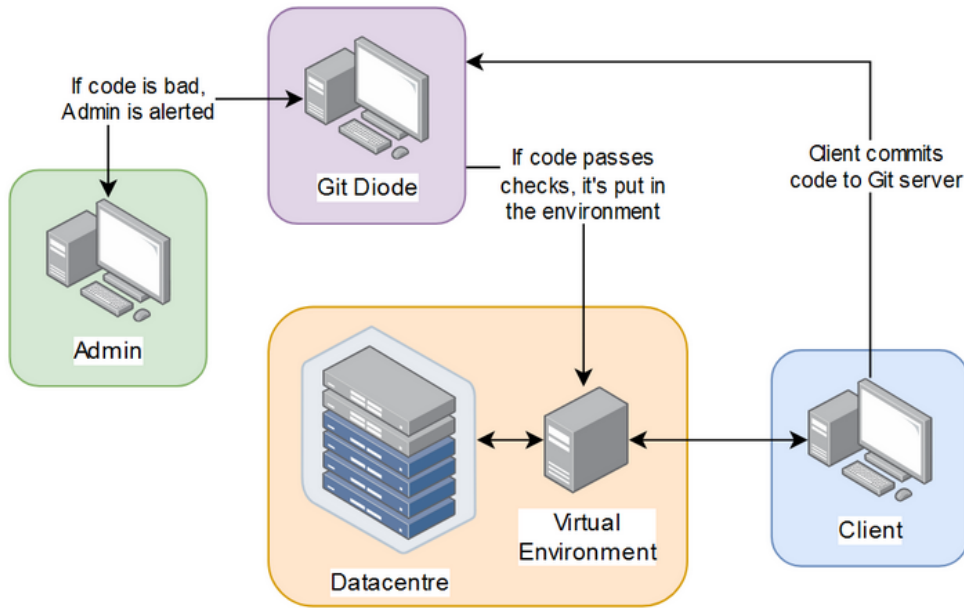


Figure 3.2: Git diode design

There are several ways this could be used depending on the threat model of that particular environment. For example, the git server could simply alert a team that there is some code that needs to be verified. However, the real power of this solution is that there are several methods that could be used to automatically audit the code.

Another benefit of using git for ingress is that all changes are logged, by nature of git. If a user attempts to commit malicious code, then deletes the code with a new commit, the overwritten code is still viewable within the history of commits. This way, researchers can be held accountable for the code they ingress.

3.2.3 Log Aggregation

Several of the proposed mitigations would generate log files, mostly logging file changes, program execution and other low level activity. With these logs being produced by several different environments across multiple research projects, it will become difficult to monitor all events without log aggregation.

There are many solutions for log monitoring and aggregation, but one free solution that is also an industry standard is Elasticsearch. This is essentially a database that allows for rapid indexing and processing of events.

Elasticsearch also has several adjacent tools that would make aggregating the logs across all of the environments trivial. Filebeat is used to send log files to the Elasticsearch database, or they can be sent to a Logstash instance, which can be used to perform some parsing on the log files before being entered into the database.

3.3 Containerisation

Virtualisation is a useful way to run applications in an isolated and safe environment, but there's some important distinctions between using virtual machines or containers to create that environment.

Virtual machines emulate the kernel and hardware, so the environment is completely isolated. This is more secure than containers, but more resource intensive and comes with additional downsides like the inability for allocated resources to be scaled in the way containers can.

Containers are lightweight environments that make use of the host operating system's kernel, but have their own file system. This means that while isolated, the applications running do share the kernel with the host and other containers.

In the case of TREs, containers could to be used to deploy software that the researchers might be using. This could be either a pre-packaged docker image or a custom built image.

Complications come from being unable to connect the environment to the internet, where pre-packaged docker images are typically pulled from, as well as it being a risk to bring anything unknown into the environment.

3.3.1 Private Docker Registry

Often docker is used for deploying generic applications such as jupyter tools and tensorboard. However, these would typically be pulled from an online docker repository, which is not possible without connection to the internet.

In order to allow researchers to bring generic docker images into the environment, a local and offline docker registry could be used to hold many generic applications. Docker images would be pulled in advance by the maintainers, then moved over to

the isolated system manually. From there, the researchers could pull any images they need without having to connect to the internet.

If a researcher required an image not on the private registry, they could just request the administrator to add it to the system as other researchers may make use of it too.

It would also be possible to implement a screening process for adding images to the private repos, mitigating supply chain attacks with poisoned images. This would not be useful for deploying researcher-specific docker images unless individual registries were maintained for each environment.

3.4 Data Egress

Data egress is the process of allowing the user to move data from within a Trusted Research Environment to outwith it. It's a fundamental necessity of TREs, but also is one of the most risk involved aspects of them.

After speaking with some maintainers of these environments, it's been determined that currently in most cases there is no restriction on which format can be egressed as long as it's human readable. For example, an excel spreadsheet of results would be fine as long as it's been manually reviewed and determined that the spreadsheet contains no sensitive information.

Also, the process of egress is currently entirely manual, where the user will place their files in a share and the maintainers of the environment will manually review the files to ensure that they are safe to be egressed. This method has worked until recently, where users are asking to be able to egress machine learning models. In this case, it's not possible to manually review the data.

Ultimately, manually reviewing the data with no formal procedure is not scalable or sustainable, and may result in missing malicious data exfiltration.

3.4.1 File Formats

As mentioned previously, there are currently no restrictions on which type of files can be egressed. As long as the file is human readable, and an administrator can read over and check it for sensitive data, it's considered safe.

The issue with this approach is that common file formats like word documents and excel spreadsheets can be used to store hidden data. These files are fundamentally a collection of xml files, and these files could be manually edited to store data as comments within the xml. Also, data obfuscation could make it so that even if the files are scanned for patient data, it wouldn't be found.

Creating a policy for which file types are allowed and which aren't would significantly improve the security of the egress process. It would become nearly impossible to hide data in files, and become a lot easier for admins to review the files for patient data.

Word documents can be represented as txt files, spreadsheets as csv and other data output as json. It should be made clear to the researchers that this is the policy, and instructions on exporting common files to minimal file formats should ideally be provided.

3.4.2 Malicious ML Model Detection

A new problem with egress is that researchers may wish to train machine learning models on patient data, then egress the trained model. These models are represented as binary blobs of data, making them impossible to manually parse for sensitive data.

There are two main potential risks here: either patient data is accidentally baked into the ML model, or patient data is deliberately hidden in a model using code. This problem is incredibly complex to solve completely, and is ultimately beyond the scope of this project.

Tools like the Adversarial Robustness Toolkit (Trusted-AI 2021) could potentially solve the problem of deliberately hiding data in ML models, but thorough testing would be needed before this could be recommended with full confidence.

3.4.3 Logging Code Execution

When it comes to steganography in images, as well as hidden data in ML models, it is very difficult to detect after the data has been encoded and hidden. One potential solution to this problem is to monitor all code that the user is executing, and catch the malicious act rather than the product of it.

Logging when code has been executed and which code was executed would create an audit trail for admins to review when people are requesting egress. As soon as someone requests egress of an ML model or images, the admins can just check if any of the code that was executed was flagged for potentially being malicious. Similar to ingress, determining malicious code is complicated, but checking for libraries related to hiding data in ML models or steganography would be a good start.

Again, logging this data isn't enough. There needs to be infrastructure and processes in place for parsing the logs, and the admins should easily be able to query for particular events or timeframes. Elasticsearch is an ideal solution for this, since it's free to use and deploy and widely used across industry.

Chapter 4

Methodology: Implementation

This chapter goes through implementing proof of concept infrastructure and code for three of the proposed solutions. Going from creating the environment used for testing to the specifics of each deployment and implementation, it should provide enough understanding that this project could be replicated.

4.1 Creating a Testing Environment

To deploy proof of concept implementations of each suggested feature, a testing environment needed to be created. This environment needed to approximately emulate a production TRE. Usually they are running a bare metal hypervisor and hosting the research environments as Virtual Machines, with other components of the infrastructure possibly being deployed as containers.

4.1.1 Hypervisor

In this case, Proxmox was chosen (*Proxmox* 2021). This is a free and open source bare metal hypervisor which provides all of the features required for this project.

In most production TRE environments, VMWare’s vSphere is used as a hypervisor, which is a closed source and commercially licensed program. Since Proxmox provides all the same functionality but without the cost of purchasing a licence, it is justifiable to use it as an alternative.

Proxmox was installed on an Intel NUC from a USB drive, and the user interface of the finished installation can be seen in figure 4.1. With this in place, virtual machines and containers could be deployed and networked together.

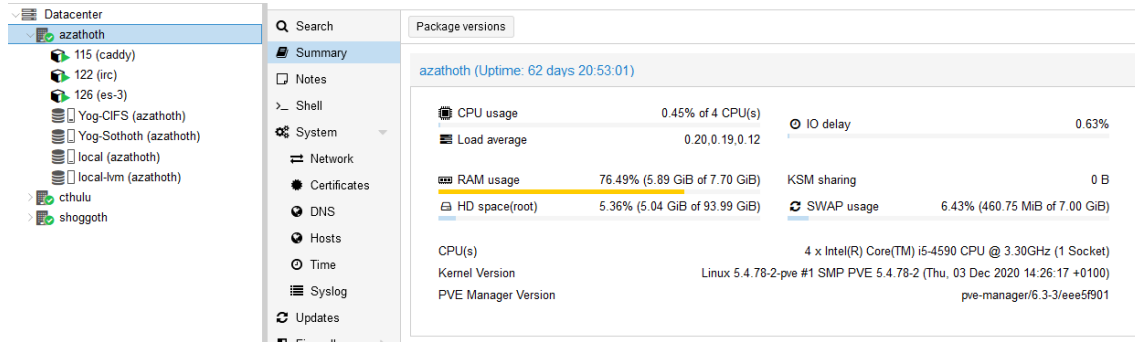


Figure 4.1: Proxmox user interface

4.1.2 Research Environments

With the hypervisor deployed, the research environment VMs could now be created. In production environments, these environments will typically be Windows hosts accessed with Remote Desktop protocols. Since Microsoft provide free trial licenses of Windows, it was possible to deploy a Windows desktop virtual machine using the ISO provided by Microsoft.

The deployed VM is shown in figure 4.2, with access being provided by Proxmox's built in Remote Desktop functionality.

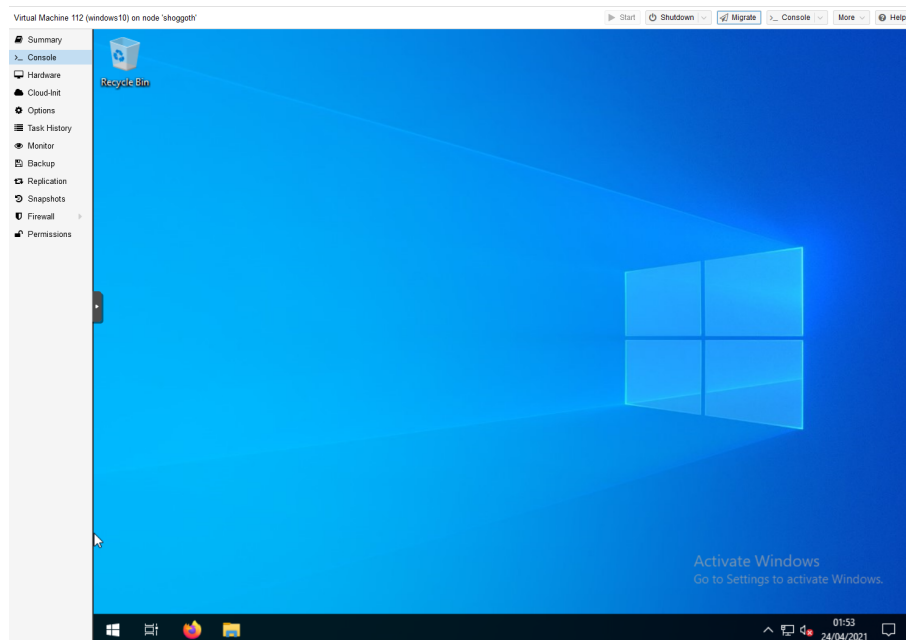


Figure 4.2: Fresh Windows install in VM

It's very likely that multiple environments like this one may need to be deployed, and it would be slow to manually create the VM each time. Proxmox has a feature that

allows for the conversion of VMs to templates, then "linked copies" of those templates can be created near instantly, allowing for the rapid deployment of arbitrary numbers of environments. In figure 4.3, three deployed environments can be seen listed.

```
root@shoggoth:~# qm list | grep "research\|NAME"
```

VMID	NAME	STATUS	MEM(MB)	BOOTDISK(GB)	PID
131	research-1	running	8192	100.00	9092
132	research-2	running	8192	100.00	11166
133	research-3	running	8192	100.00	11242

Figure 4.3: Three template copies deployed

4.1.3 Networking and Firewalls

These environments shouldn't be able to communicate with the internet, but should be allowed to have limited communications with certain local hosts. In a real environment, the firewall rules would be as granular as possible, allowing the bare minimum traffic that is required for internal services to communicate and function.

Since more granular firewall rules wouldn't be possible until the services to be communicated with were being tested, some basic firewall rules that prevented internet access but allowed LAN communication were deployed.

```
[RULES]

# allow traffic on local network
IN ACCEPT -source 192.168.1.1-192.168.1.255 -log nolog
OUT ACCEPT -dest 192.168.1.1-192.168.1.255 -log nolog

# drop all other traffic
IN DROP -log nolog
OUT DROP -log nolog
```

Figure 4.4: Some basic firewall rules

4.2 Git Diode

4.2.1 Ingress Share

The first step was to create a network share on each research environment that could be accessed by the git diode to place the code. Figure 4.5 shows that the folder Ingress was configured as a network share, accessible at `\\RESEARCH-1\\Ingress`.

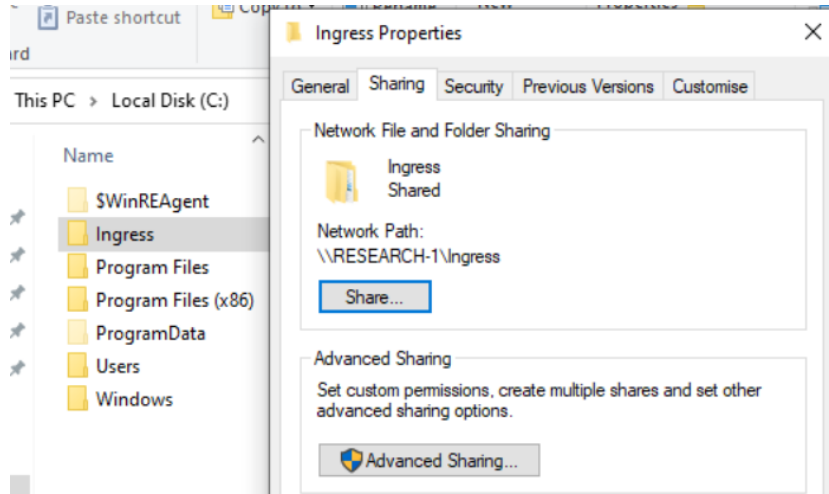


Figure 4.5: Ingress share configured

Since the git diode would be a linux host, the mount cifs tools had to be installed in order to mount the share. In figure 4.6, the folder has been successfully mounted as a cifs share and the contents can be read or written to.

```
sam@1az:~/ingress$ sudo mount.cifs -o credentials=/etc/win-credentials //RESEARCH-1/Ingress /mnt/ingress
sam@1az:~/ingress$ findmnt -D | grep Ingress
//RESEARCH-1/Ingress cifs 99.5G 26G 73.5G 26% /mnt/ingress
sam@1az:~/ingress$ ls /mnt/ingress/
test-code
```

Figure 4.6: Windows share mounted

Figure 4.6 also shows that a file with stored credentials, `/etc/win-credentials`, was used to authenticate with the Windows host.

4.2.2 Scanning Script

In order to determine whether code should be allowed into the environment or not, a script needed to be created. Since the GitLab CI tools needed a bash script to execute, as well as the script needing to run several commands, bash was chosen as the language to implement the script in.

The first part of the script is the function for checking if the code should be allowed. Scanning for malicious code is a very complex problem, and out of scope of this proof of concept. A placeholder function was created that simply greps for the word 'bad' within the files and returns true if it's found, otherwise false:

```
function check_code() {
    if grep -rq "bad" $1; then
        return 1
    else
        return 0
    fi
}
```

Of course in reality the scanning process would be much more involved, and this function can easily be replaced with a more complex scanning solution.

The second part of the script deals with the ingress process. If the checks are passed, the code can be automatically ingressed into the environment. The share of each environment is mounted at `/mnt/ingress/[projectname]`, so all that needs to be done - if the function returns true - is copy the code into the share:

```
# scan code
if check_code .; then
    echo 'code is fine'
    cp -r . /mnt/ingress/$1/
    echo 'code transferred to ingress share'
```

The project name is passed in as a parameter to the script. Finally, if the check fails, the code is flagged for review and not ingressed. This is covered in more detail in the ELK section. The full finished script can be seen at appendix A.1.

4.2.3 GitLab Deployment and CI Pipeline

In order to use a content integration pipeline, a local GitLab instance needed to be deployed. GitLab was chosen as it's a widely used open source project with comprehensive content integration features, and could easily be self hosted.

This was a fairly straightforward deployment, using a Debian 10 container on Proxmox as a host. After initial configuration was complete, repositories for each research project could be created, shown in figure 4.7

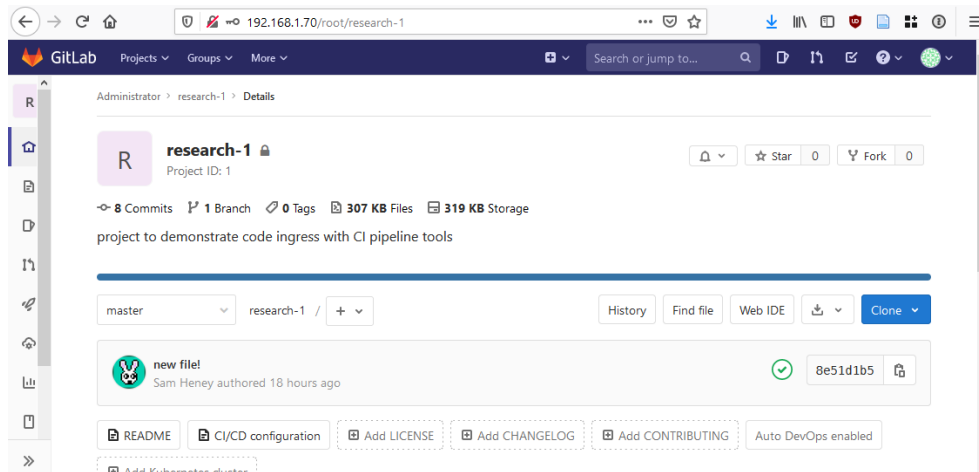


Figure 4.7: Repository for a research project

In this case, the repository was created and managed with the root account, but in a real environment different accounts would need to be created for each user. The permissions would also need to be managed so that the user could only use their respective project's repository.

GitLab content integration works by creating jobs that execute at certain points of deployment. These jobs are defined by a config file within the repo, then are executed by a Runner service.

The Runner service needs to be deployed on a separate host to the GitLab instance, so another container was created and the Runner service configured and installed.

Once deployed, the service allows Runners to be created. Each runner has to be linked to the GitLab instance by providing the URL and registration token during its creation. The created runner can be seen in figure 4.8.

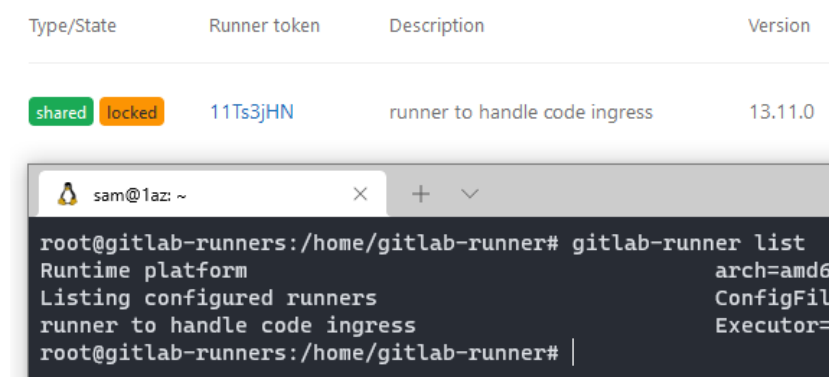


Figure 4.8: Runner shown in the web interface and in the runner list on the server

The type of this runner is shared, which means it can be used by multiple different projects. This is again a simplification for the proof of concept, but in a production environment a different runner could be created for each project, and set to specific, meaning only that project could use it.

With this in place, jobs can be executed. To create a job for code ingress, first the ingress script needed to be added to the path, so it was placed in `/usr/local/bin/`. Then, a `.gitlab-ci.yml` file could be added to the repo, to describe the ingress job:

```
job:
  script:
    - ingress research-1
```

Now any commits to the repository that are pushed to the GitLab instance will trigger this job, causing the `ingress` script to execute for this project.

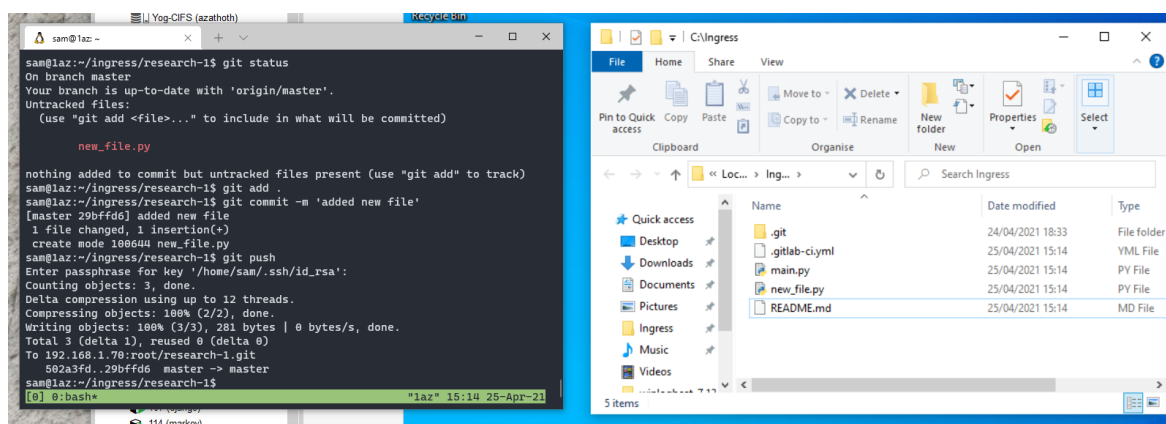


Figure 4.9: New file being added to the project, pushed, then ingressed

4.3 Activity Monitoring and Log Aggregation

4.3.1 Logging Process Execution

By default on Windows systems, many different system activities and events are logged to event viewer. This gives a comprehensive understanding of what has happened on the system, and is perfect for creating an audit trail.

However, by default process creation and termination are not logged. This needs to be enabled by changing the Group Policy. In a production environment, this would

simply be a case of changing the policy on the Active Directory server and propagating the changes to each environment. Since no AD server was set up for this proof of concept, the policies had to be changed on each environment using the Local Group Policy Editor, seen in figure 4.10.

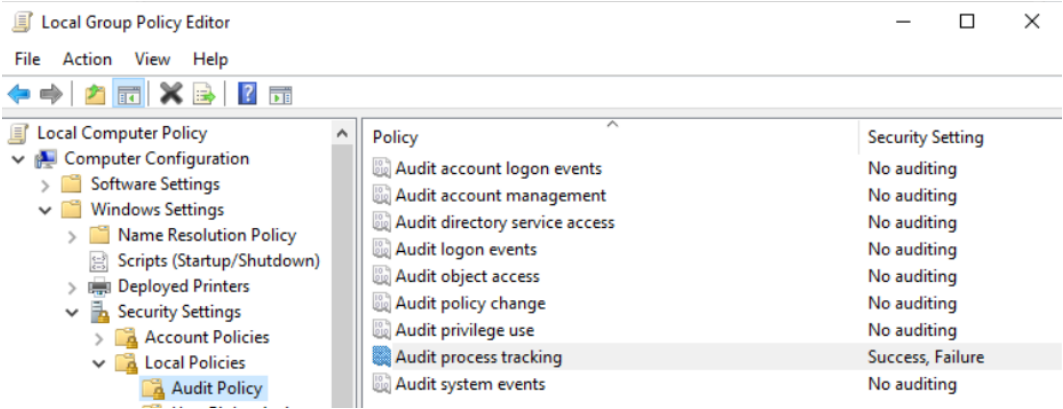


Figure 4.10: Process tracking enabled

With this enabled, any new processes that are created or terminated are logged to event viewer. A sample of these logs is shown in figure 4.11.

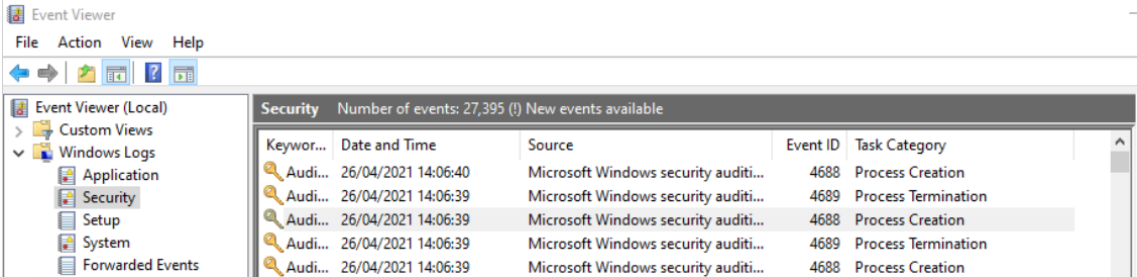


Figure 4.11: Sample of process logs

Each logged event contains detailed information, including who executed the process, what the process was, what domain it was executed in and more. This is shown in figure 4.12.

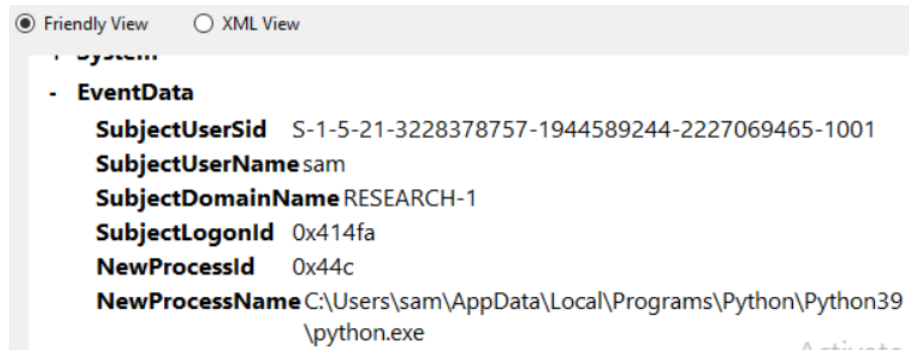


Figure 4.12: Sample of the detail of each logged event

However, this only gave an overview of processes. There are more attributes of user activity that are still not covered by the default windows logging, namely file creation. To get this information, the tool **Sysmon** was used (Russovich 2021). This is a general purpose tool that generates logs for many events that are missed by the default windows logging.

Using **Sysmon** logging the process information, and using all of the other logs that Windows generates by default, should be sufficient to create a comprehensive trail of activity for any given time frame. This process was done across three different environments for testing purposes.

4.3.2 Deploying ELK Stack

ELK stands for Elasticsearch, Logstash and Kibana. These three components work together, serving different functions in the pipeline from log generation to aggregation and visualisation.

Elasticsearch is the first and most important component, since it can work without the other two but they can't work without it. It's essentially a database optimised for searching, ideal for filtering for specific things or events in log files.

Logstash is the ingestion pipeline, where log sources send their data to be pre-processed before it's sent to Elasticsearch. It has many powerful tools for parsing and generating information about log files.

Finally, there's Kibana. This is a web application that provides a visual interface to the Elasticsearch database. It also allows the rapid construction of visualisation dashboards.

First, Elasticsearch was deployed. In this case, three Elasticsearch nodes were deployed across three different physical hosts. This was to demonstrate the clustering functionality of Elasticsearch and how it can be made to scale easily, and since the amount of traffic it will be ingesting is significant, this ensures that no bandwidth issues will prevent ingestion.

The configuration of each host was simple: each node just had to be made aware of the other nodes in the network and Elasticsearch handles the clustering automatically. Figure 4.13 shows the full config file for **es-1**, the first node.

```
# ===== Elasticsearch Configuration =====
#
# The primary way of configuring a node is via this file. This template lists
# the most important settings you may want to configure for a production cluster.
#
# ----- Cluster -----
cluster.name: dunwich-elastic
# ----- Node -----
node.name: es-1
# ----- Paths -----
path.data: /data/elasticsearch
path.logs: /var/log/elasticsearch
# ----- Network -----
network.host: 0.0.0.0
# ----- Discovery -----
discovery.seed_hosts:
  - 192.168.1.80
  - 192.168.1.81
  - 192.168.1.82
# Bootstrap the cluster using an initial set of master-eligible nodes:
cluster.initial_master_nodes:
  - es-1
  - es-2
```

Figure 4.13: es-1 node configuration

With this configuration in place and the other nodes configured the same, the Elasticsearch service could be started. After allowing some time for the nodes to automatically configure the cluster, any of the hosts could be queried for data.

Figure 4.14 shows a query being sent to **es-1** requesting the ip, master status and name of each node. The response shows that each node has been successfully registered, and that **es-1** has been made the master node.

```
sam@1az:~$ curl http://192.168.1.80:9200/_cat/nodes?h=ip,m,name
192.168.1.82 - es-3
192.168.1.80 * es-1
192.168.1.81 - es-2
```

Figure 4.14: Elasticsearch nodes deployed

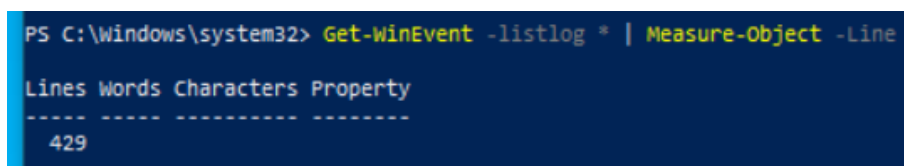
With the Elasticsearch cluster working, Logstash and Kibana could be deployed. These were each deployed on their own separate containers, and each had minimal configuration that just involved pointing them to the three Elasticsearch nodes.

4.3.3 Sending Logs

Now that the ELK stack was deployed, data could be sent to it. There are many different ways to send data to Elasticsearch but one of the primary mechanisms are **beats**, which are lightweight applications that send data from systems.

In the case of a Windows system, there is a beat called **winlogbeat** (Elastic 2021). This is designed to read the event viewer log format and send them to either Logstash or straight to Elasticsearch.

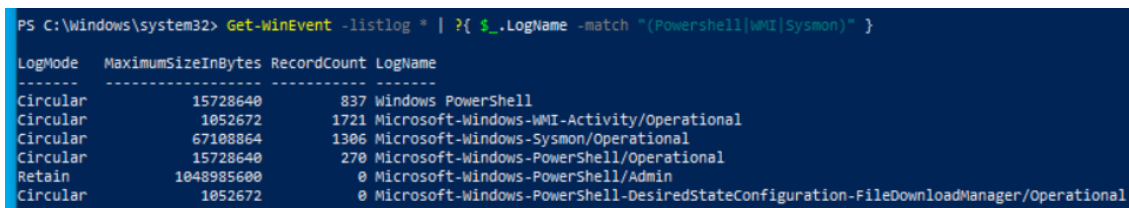
After downloading the application, but before activating it, the logs that should be sent need to be configured. As figure 4.15 shows, there are hundreds of logging sources available, and far too much data to process meaningfully available for use, so key sources would need to be determined.



```
PS C:\Windows\system32> Get-WinEvent -listlog * | Measure-Object -Line
Lines Words Characters Property
-----
429
```

Figure 4.15: Count of log sources

Aside from the default Windows logging paths, some other important information comes from Powershell events, Sysmon logs and Windows Management Instrumentation (WMI) information. In order to find which log sources would provide this information, Powershell was used to search through all the log sources for those whose name contained Powershell, WMI or Sysmon.



```
PS C:\Windows\system32> Get-WinEvent -listlog * | ?{ $_.LogName -match "(PowerShell|WMI|Sysmon)" }
LogMode MaximumSizeInBytes RecordCount LogName
-----
Circular 15728640 837 Windows PowerShell
Circular 1052672 1721 Microsoft-Windows-WMI-Activity/Operational
Circular 67108864 1306 Microsoft-Windows-Sysmon/Operational
Circular 15728640 270 Microsoft-Windows-PowerShell/Operational
Retain 1048985600 0 Microsoft-Windows-PowerShell/Admin
Circular 1052672 0 Microsoft-Windows-PowerShell-DesiredStateConfiguration-FileDownloadManager/Operational
```

Figure 4.16: Relevant log sources after searching

Winlogbeat also comes with several scripts to parse these extra logs. For example, figure 4.17 shows the final configuration for Powershell events and Sysmon events, where only some Powershell event ids are sent and both are passed through a preprocessing script supplied with winlogbeat.

```
- name: Microsoft-Windows-Sysmon/Operational
  processors:
    - script:
        lang: javascript
        id: sysmon
        file: ${path.home}/module/sysmon/config/winlogbeat-sysmon.js

- name: Windows PowerShell
  event_id: 400, 403, 600, 800
  processors:
    - script:
        lang: javascript
        id: powershell
        file: ${path.home}/module/powershell/config/winlogbeat-powershell.js
```

Figure 4.17: Winlogbeat config

With winlogbeat configured, the service can be started with powershell's **Start-Service** cmdlet. This will immediately start sending logs to the Elasticsearch instance, which can be confirmed by checking for the winlogbeat index in the list of indexes, as seen in figure 4.18.

```
sam@laz:~$ curl http://192.168.1.80:9200/_cat/indices?pretty -s | grep win
green open winlogbeat-7.12.0-2021.04.25-000001 huc1HdqMQDWR-1BN7Tgjkg 1 1 45835
```

Figure 4.18: Winlogbeat index

Once again, this configuration was applied across all three research environments.

4.3.4 Visualisation

Now that the data is being sent to and ingressed into the Elasticsearch database, Kibana can be used for visualisation. There are a few different options when it comes to visualising the events.

Firstly, the "Discover" page allows the user to manually parse all of the events that have been recieved. Figure 4.19 shows this page, and it's immediately clear that this is not a feasible way to parse though the thousands of events in a meaningful way. It is however useful for finding interesting attributes of the data to sort by.



Figure 4.19: Kibana Discovery page

Kibana also provides tools to manually build a dashboard of events. One of these is interface that allows the construction of "lenses", which essentially allows for click-and-drag construction of real time graphs and different ways to break down the data.

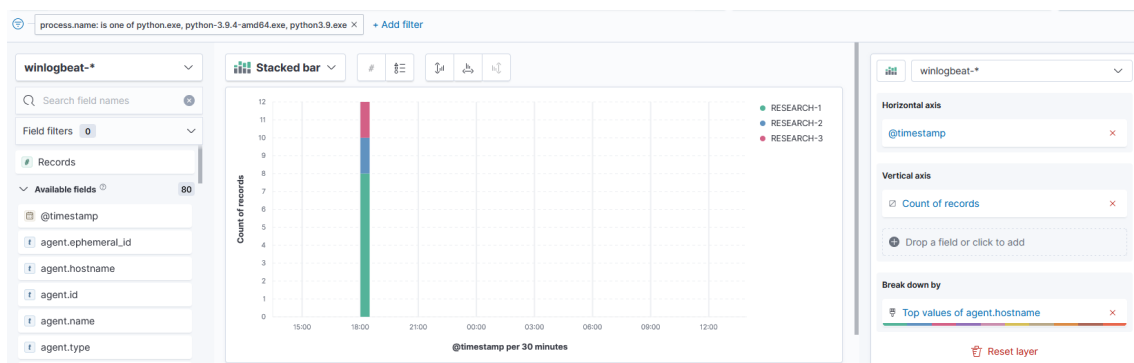


Figure 4.20: Kibana Python executions lens

These lenses can then be combined to form a Dashboard. There are many different possibilities with such a powerful system, and this would also be a good way to implement "alert" events, for example if code ingress fails.

Figure A.1 in appendix A.2 shows an example of a dashboard that could be used to monitor the research environments, with three different lenses. These lenses show python executions, total events, and the breakdown of events across environments. Figure A.2 shows the dashboard after filtering only for events that are children of powershell.

Beyond the custom dashboards, Kibana has a pre-made "Events" section that allows the user to view aggregated events from all monitored endpoints, as well as a graph showing breakdowns of which type each event is. This can be seen in figure 4.21.

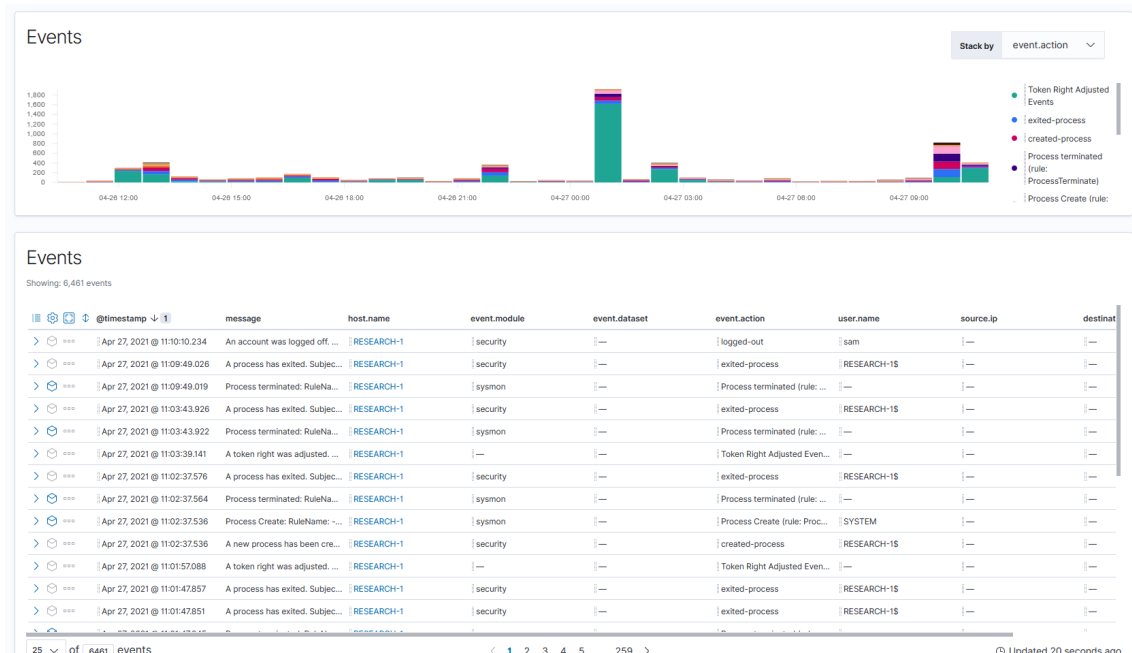


Figure 4.21: Process list

This is a great way to filter for events that might look suspicious, for example processes that are spawned from python. The user should not be launching applications via python, so this could be considered suspicious activity. Kibana allows for the further analysis of suspicious processes, allowing the user to examine process trees of an event that might be flagged as suspicious.

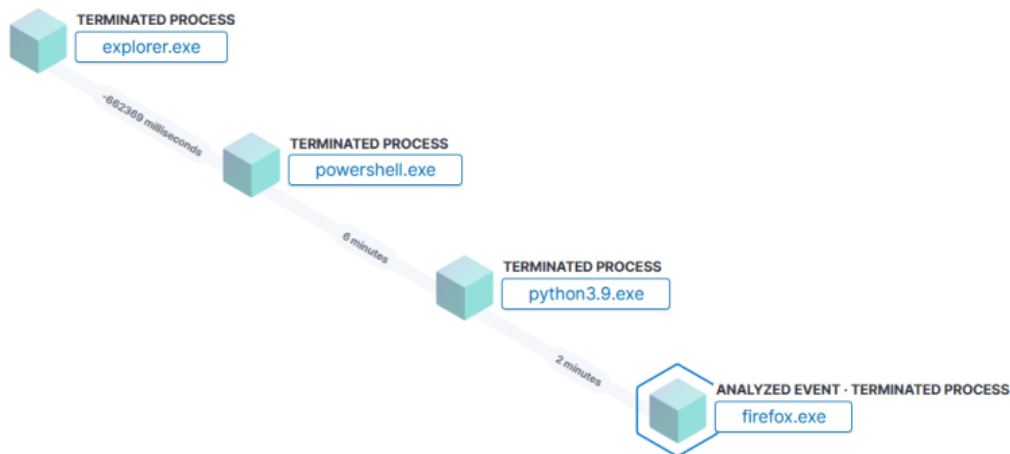


Figure 4.22: Python starting a Firefox process

Figure 4.22 shows the example that was mentioned above, where a firefox process was started from Python and run for two minutes before terminating. This demonstrates how useful this pipeline is for creating an audit trail for responding to any potential incidents of malicious activity.

4.4 Automated Egress Checks

4.4.1 Egress Share

Similar to the Git Diode, first a share needed to be created on the research environments and mounted on the container where the Egress process will be managed. The full process for creating and mounting the share can be read in section 4.2.1.

4.4.2 Automatic Scanning

Whenever new files are added to the **Egress** share, they should be checked and verified in various ways. These checks may be intensive, so it's not reasonable to be constantly running the checks at all times. Instead, the checks should only be executed if there have been changes either in the file contents or if a file has been added/removed.

There are several tools that make use of underlying linux filesystem features that would serve this purpose, such as `entr` and `inotify`, but it was found that they didn't

work. It's likely that this is due to the mounted share being from a windows host, but regardless, an alternative was necessary.

Since changes to the filesystem would need to be manually tracked, a method for getting information about each file on the share was required. The tool `du` was decided on, since when combined with `find` it could easily get file size information of all files and folders on the share. Figure 4.23 shows what the output looks like.

```
root@egress:~# du -ab $(find -L /mnt/egress/research-1/)
179      /mnt/egress/research-1/B00P.jpg
179      /mnt/egress/research-1/B00P.png
179      /mnt/egress/research-1/B00P.rtf
43096    /mnt/egress/research-1/SDFADF/DSFD.txt
43096    /mnt/egress/research-1/SDFADF
13       /mnt/egress/research-1/TEST.txt
43646    /mnt/egress/research-1/
```

Figure 4.23: Sample du output

With a command that got a full recursive overview of a directory, this output could just be stored in a file, then when the command is run again, that output compared to the cached one. If it was different, a change has been made. This is trivial in bash scripting, where the command `cmp` was used to compare the cached file contents to the stdin piped in from the `du` command:

```
if ! $(du -ab $(find -L $scan_path) | cmp -s "$cache_file"); then
    echo "changes detected"
    # perform checks on egress dir here
    du -ab $(find -L $scan_path) > "$cache_file"
else
    echo "no changes detected"
fi
```

The comment indicates where the call to the egress checks should be placed. Figure 4.24 shows the scan script functioning correctly, detecting when a file has been changed one time before again finding no changes.

```
root@egress:~# ./scan
no changes detected
root@egress:~# touch /mnt/egress/research-1/new.txt
root@egress:~# ./scan
changes detected
root@egress:~# ./scan
no changes detected
```

Figure 4.24: Scan script working

4.4.3 Verifying File Type

Files give two main clues as to what the filetype will be: the extension of the file and the file signature, aka "magic bytes", of the file. The extension is simply the word at the end of the filename, like how `document.docx` would indicate a Microsoft Word file.

The file signature is the first few bytes of the file, which typically are unique to each file format. For example, the bytes `504B0304` represents the zip file format, which is what the docx type is based on, so this implies that the file could be a docx file (Kessler 2021).

There are several places on the internet where a comprehensive list of file signatures can be found, such as Gary Kessler's file signatures table, a site that started in 2002 that is still being updated to this day (Kessler 2021).

Making use of this table, a json array of file types, their signatures, and some other information could be compiled. A sample of this array can be seen in figure 4.25.

```
{
  "type": "document", "extension": "pdf", "offset": 0, "signature": ["25 50 44 46"]},
  {"type": "document", "extension": "rtf", "offset": 0, "signature": ["7B 5C 72 74 66 31"]},
  {"type": "document", "extension": "epub", "offset": 0, "signature": ["50 4B 03 04"]},
  {"type": "document", "extension": "xml", "offset": 2, "signature": ["78 6D 6C"]},
  {"type": "archive", "extension": "7z", "offset": 0, "signature": ["37 7A BC AF 27 1C"]},
  {"type": "archive", "extension": "rar", "offset": 0, "signature": ["52 61 72 21 1A 07 00", "52 61 72 21 1A 07 01 00"]},
  {"type": "archive", "extension": "tar.z", "offset": 0, "signature": ["1F 9D", "1F A0"]},
  {"type": "archive", "extension": "gz", "offset": 0, "signature": ["1F 8B 08"]},
  {"type": "archive", "extension": "zip", "offset": 0, "signature": ["50 4B 03 04", "50 4B 05 06", "50 4B 07 08"]},
}
```

Figure 4.25: Sample of the type list json

With this list assembled, each file can be checked for their extension and signature. This way, misrepresentations of the real file type can be checked for, while also ensuring that the file is of a type that is allowed for egress.

Python was chosen as the language to implement the type checking with. First, since each file in the given share would need to be checked, a recursive method of finding each file within the share and its subfolders was needed. Since Python was being used, the `os.walk()` method could be used:

```
if __name__ == '__main__':
    # for each folder within the share
    for root, dirs, files in os.walk(sys.argv[1]):
        for name in files:
            print(os.path.join(root, name))
            # pass the first 128 bytes and the
            # filename to the "check" function
            with open(os.path.join(root, name), 'rb') as f:
                check(f.read(128), os.path.splitext(name))
```

This loop calls the `check()` function on each first 128 bytes of a file and its file name. This function needed to take these parameters and check the list of file type information to see if the file type can be determined. Firstly, the byte stream needed to be converted to a hex digest delimited by a space, as it is in the json array:

```
def check(b, ext):  
    stream = b.hex(' ')
```

Fortunately Python 3.8 introduced a new feature to the `.hex()` method for byte strings which allows for the first parameter to set the delimiter of the hex string. Unfortunately, since Python 3.7 is the default on Debian, which is what the container is running on, the newer version had to be compiled and installed from source to make use of the method.

Next, the list of types could be iterated over and the file checked. This involved splicing the file signature out of the byte stream based on the length of the file signature. If the file signature matches the one from the json list, the file extension can be checked to ensure it's consistent with the detected type:

```
for t in types:  
    for signature in t['signature']:  
        offset = t['offset'] * 2 + t['offset']  
        if signature == stream[offset:len(signature) + offset].upper():  
            # file type has been found  
            if (t['extension'] != ext[1][1:]):  
                print('extension mismatch detected!')
```

The final result of the bash script calling the Python program when it detects changes is shown in figure 4.26. This shows some files that have the extension of image files but are detected to be "document" type, which is flagged as a mismatch. In the case of plain text, no type is detected, and this is reflected in the output as well. The final full code can be found in appendix A.3.

```

root@egress:~# ./scan.sh
no changes detected
root@egress:~# cp /mnt/egress/research-1/B00P.rtf /mnt/egress/research-1/B00P.jpg
root@egress:~# ./scan.sh
changes detected
/mnt/egress/research-1/B00P.jpg
file type is document
extension mismatch detected!
/mnt/egress/research-1/B00P.png
file type is document
extension mismatch detected!
/mnt/egress/research-1/B00P.rtf
file type is document
/mnt/egress/research-1/TEST.txt
no file type detected, probably plain text
/mnt/egress/research-1/SDFADF/DSFD.txt
no file type detected, probably plain text
root@egress:~# |

```

Figure 4.26: Scanning changed files

The final step to using this solution is to make the script run regularly. This could be done trivially with a cron job, or by creating a systemd service that automatically runs the script at a regular intervals. In this case a cron job was configured, as shown in figure 4.27

```

# m h dom mon dow  command
* * * * * /root/scan /mnt/egress/research-1/

```

Figure 4.27: Cron job

Results

This chapter provides an overview of all infrastructure deployed for this project, as well as brief summaries of each section of the network. It also illustrates how these components might fit into real TRE infrastructure.

5.1 Network Diagram

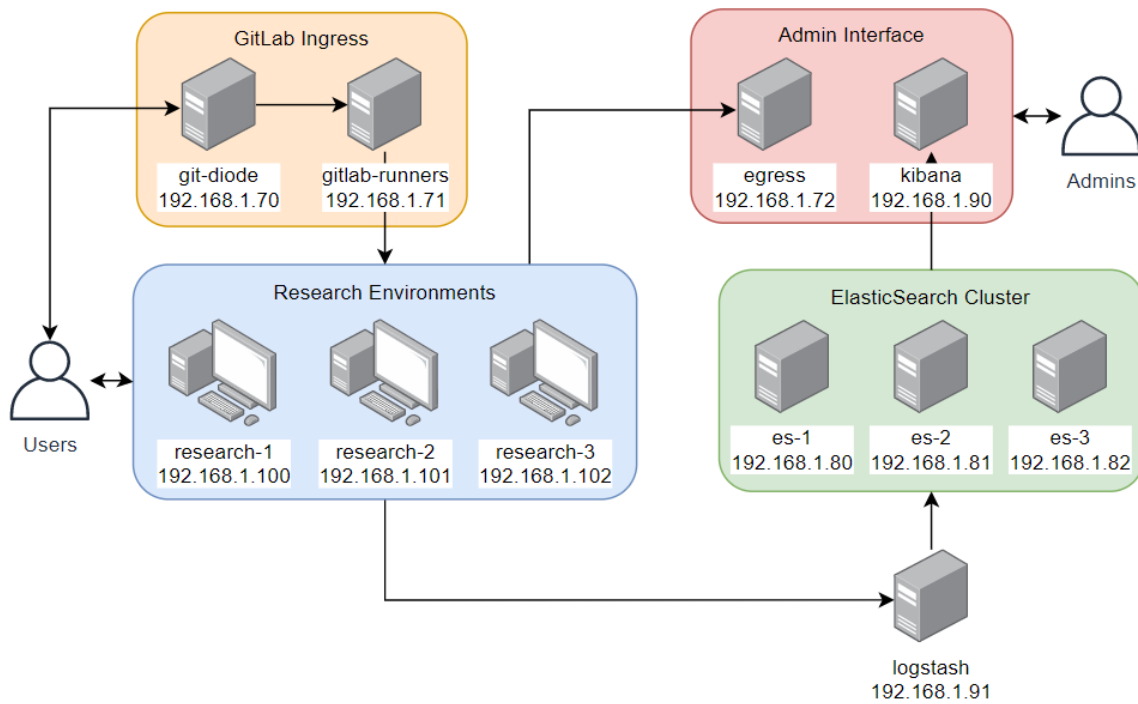


Figure 5.1: Diagram of final infrastructure

5.2 Topology Overview

5.2.1 Research Environments

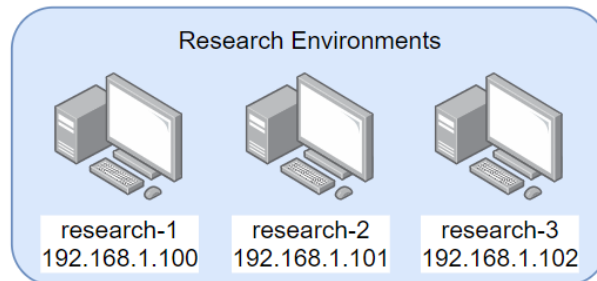


Figure 5.2: Research Environments

This part of the project was completely successful. Three Windows 10 environments were deployed and used for testing, but making use of the templating method outlined in section 4.1.2 would have allowed for trivial deployment of new hosts if it had been necessary.

5.2.2 GitLab Ingress

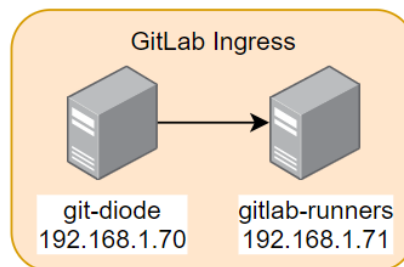


Figure 5.3: GitLab Ingress

These hosts are for the ingress system described in section 4.2. Figure 5.1 shows how the `git-diode` host would be the other user interface to the system, with the `gitlab-runners` host sending code into the research environments. The configuration described in the design stage was fully implemented, and the system is fully functional.

5.2.3 Elasticsearch Cluster

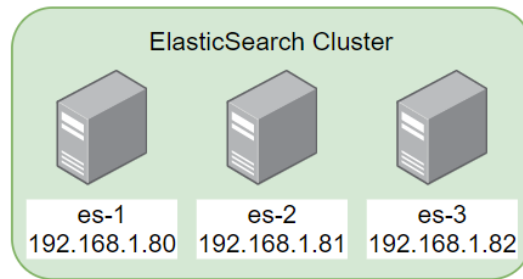


Figure 5.4: Elasticsearch Cluster

These hosts are functionally identical, working together to form the single Elasticsearch cluster that can be interfaced with by querying any of them. The logs from all research environments are received, indexed and stored. Kibana can then query the cluster to access those logs. Once again, this was a complete success, and the full intended functionality was implemented.

5.2.4 Admin Interface

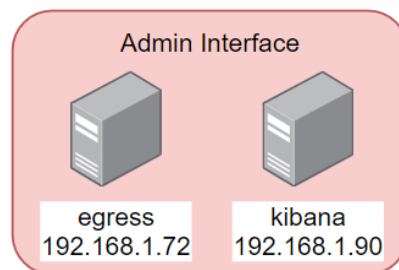


Figure 5.5: Admin Interface

Finally, there's the two hosts an Admin could use to make use of this infrastructure: **egress** and **kibana**. All information about the infrastructure is accessible via the Kibana interface, while the egress share has access to the files on each research environment that the researchers are requesting to be egressed. This section wasn't fully completed, as ideally the output of the "egress" host would be completely accessible within Kibana. Due to time constraints, this wasn't implemented.

Chapter 6

Discussion

This chapter is an in depth reflection on the whole project. It explores how each part informed and determined the following parts, as well as summarising the positives and negatives of each proposed solution. The work is also related back to the aim and research questions, with the success of the project in relation to each of these addressed.

6.1 Git Diode

This system was designed to solve the problem of code ingress. Gitlab’s content integration tooling was used to allow researchers to ingress code into their TRE.

All of the risks identified in the design stage are successfully addressed and mitigated by this solution. Essentially any means of auditing code would, but this infrastructure also provides scalable ways for automating detection methods for bad code, improving efficiency without putting security at any further risk.

The main advantage of this solution is that it provides a familiar interface to developers. Git is a very commonly used tool for development, and making that the way the researchers ingress code should be comfortable. If the code is already in a Git repository, it can easily be pushed to the Gitlab instance.

Also, the system is easily scalable. Different repositories and accounts can trivially be added for each project, and these repositories are only visible to those who are given access. The runner service can be run on multiple containers if traffic became too high for a single instance to handle.

In terms of integration with existing TRE infrastructure, this wouldn’t interfere with any other type of service. Since ingress is already done manually, whatever location on the research environments is normally used can be used as the ingress share to be targeted by the job.

It may be better to configure it with an intermediary storage server that hosts the ingress shares for each environment, then the environments mount the shares on that server. This way, the Gitlab runner jobs never touch any environments with access to the sensitive data, reducing the chance of a data breach if a bad actor gained access to the Gitlab server somehow.

In this implementation, the actual method of scanning the code for badness was not implemented as it is considered beyond the scope of this project. There are many different possible ways to check if code might be bad, but one simple way in a case like this would be to check the libraries in use for any that could be used for obfuscation, steganography or similarly malicious techniques.

One negative aspect of this implementation is that it requires a file to be placed inside the git repository to trigger the CI pipeline job. This is because usually CI is just used for building the project, which is most likely going to change over time as the project develops. This means that technically if the Gitlab CI file in the repository was maliciously altered, a user could execute arbitrary commands.

A mitigation for this could be to store a hash of the CI file and then to check the hash before execution, but even then the user still has the power to break it. Ultimately there should be a better way of doing this that requires no configuration from files within the users control.

In summary, this component is a good proof of concept as a solution to the problem of code ingress, but some aspects of it need some more thought before it could realistically be used in a production environment.

6.2 Activity Monitoring and Log Aggregation

This system was designed to solve the problem of log aggregation and analysis, in order to enable administrators to gain a better insight into user activity within the TREs. The solution has multiple components: using Sysmon to get more detailed logs, Winlogbeat to send the logs back, and the ELK stack to ingest, index and visualise the logs.

The choice of technologies to use were heavily influenced by the findings of the literature review, where multiple papers vouched for the effectiveness of the ELK stack. This ended up proving to be a good decision, as the ELK stack has been useful

in a number of different ways for this project.

Elasticsearch as a database is very powerful, and the easy to use querying syntax is great for quickly testing different configurations and indexes. There is no risk of performance issues since it was designed to be scalable.

If more research environments were introduced with different operating systems, such as Ubuntu, this system could easily adapt to that. The ideas of tracking process parent child relationships apply both to Linux and Windows, and there are existing solutions like Winlogbeat for Linux systems. If anything, implementing similar endpoint logging on a Linux host would be easier.

The fact that ELK is free and open source is ideal for saving money in a potentially budget constricted environment. The alternatives, as discussed in the literature review, are very expensive.

Kibana proved to be a really effective and intuitive way to interact with the vast amount of data from the logs being ingested. Visualisations were trivial to craft, and the ability to create a custom dashboard has endless potential. The "Hosts" section is also extremely useful, providing methods to analyse parent-child relationships of processes and flag any potentially anomalous events.

In terms of how this would fit into existing infrastructure, it depends on how the admins would deploy it. It's possible to deploy the entire ELK stack on one host, but then it's significantly more difficult to scale since it would need to be reconfigured as a cluster. Deploying the full stack took a significant amount of work, and it may be that TRE companies don't have the resources to dedicate to it.

Another issue with ELK is that all of the alerts and triage triggers need to be manually constructed. This requires a TRE org to dedicate time and resources to creating detection rules and methods, which is a difficult process. Since this situation is so unique, it's unlikely pre-made solutions for detecting anomalous events in the context of a TRE exist, so even if the org were to invest in a solution like splunk that comes with many pre-built detection methods, it might not have the right tools for a TRE scenario.

In summary, the ELK stack is a completely viable solution for a TRE organisation to implement SIEM functionality and endpoint monitoring, which could help mitigate many different risks. However, since Elastic don't provide support, it would be on the engineers to create sufficient rules and tooling to detect the malicious activity.

6.3 Automated Egress Checks

The last implemented component, this was designed to solve the problem of data egress. This would allow researchers to get data they have created in the environment out of it, and is a necessary component of a TRE. Currently the egress methods are entirely manual, with the researchers notifying the admins that they would like to egress some data, and the admins manually inspecting it for data that shouldn't leave the environments.

Many of the risks identified in the design section are addressed by this solution. Limiting the file types that can be ex-filtrated to just plain data vastly reduces the risk of steganographic exfiltration. Similarly, obfuscated data should be easier to identify when all the data in the file is visible.

The automation of detecting when files need to be egressed and notifying the admins would significantly improve the efficiency of their workflow. Rather than having to rely on communication from researchers, they can just use the aggregated notifications from when files are added to any egress share to check the files.

Also, the automation of file type detection would easily enable the implementation of a file type allow-list, where complex files are disallowed and the users are immediately notified that egress of those will not be possible. This was not implemented in the POC, but could trivially be added.

There are however some downsides to this solution. Firstly, it doesn't address the problem of egressing images. However, theoretically any image based results would most likely be graphs of data, so the underlying data could just be exported and that could be enforced. It's unlikely that exporting images is an absolute necessity.

It also doesn't address the issue of exporting machine learning models. As mentioned in the design chapter, these are just blobs of data, and it's very difficult to determine whether they contain hidden data. However, combining the logging and monitoring stack with the code ingress solution should make it difficult for a malicious actor to bring in code that would produce such an image, so this risk is partially mitigated by that.

Implementing the ML model scanning toolkit into the egress share scanning process is possible, but would need thorough testing to ensure it can reliably detect hidden data.

When it comes to integrating this with existing TRE infrastructure, much like the other components, it shouldn't interfere or replace any existing tools, but just be a further addition. These scripts can be deployed on a container and the shares of each TRE mounted trivially, so deployment won't be an issue. One risk is that if this host is not made secure, an attacker who has breached the network and gained access to it could use this to at best read all egressed files, and at worst pivot into the research environments themselves.

In summary, this solution will enhance the workflow of the TRE admins significantly, and by adding several automated checks can be used to enforce certain restrictions on egress automatically. However, some egress related problems still remain unsolved, and this is not a complete automation of egress.

6.4 Evaluation of Success

6.4.1 Aim

The aim of this project was to identify some missing security features of existing Trusted Research Environments, discuss potential solutions, create proof-of-concept implementations and evaluate their potential.

Through the literature review and design stage of this project, missing security features were identified and potential solutions were discussed. As mentioned in the introduction, it would be impossible to address every security problem and for each solution to be perfect, but these chapters do provide a sufficiently comprehensive coverage of several ideas that can be used to improve the security of TREs. These range from policy changes like limiting file types for egress, to practical solutions like deploying an ELK stack.

The implementation chapter demonstrates three proof of concept implementations of the ideas discussed. Further than that however, each POC was designed and implemented to work together, so that at the end of the project the whole system could be deployed as a pseudo-TRE.

The holistic nature of the POCs was useful in the evaluation of each solution throughout this discussion. While the solutions were not perfect and several downsides were identified with each, they are still successful in mitigating many of the risks

identified, and they were designed in such a way that should make it clear how they might fit into existing TRE infrastructure.

6.4.2 Research Question 1

What are current TRE implementations lacking?

This question was addressed by the literature review and by the design phase. Existing TRE infrastructure was evaluated and various risks were identified that did not seem to be sufficiently addressed. Various solutions were designed to mitigate these risks.

6.4.3 Research Question 2

Of those lacking features, how can these be implemented securely?

This was answered across both methodology chapters. In design, each suggested feature was presented alongside any potential security implications that deploying them in real infrastructure may have. Some solutions like code ingress still have some security flaws that were addressed in the discussion, but mitigations for those were also discussed.

6.4.4 Research Question 3

How can existing technologies help improve security of TRE
infrastructure?

Finally, this was answered by the literature review research, but also through the design and implementation chapters. The ELK stack is a technology that was not heavily mentioned in the design chapter as it seemed tangential to the other problems being solved. However, throughout implementing each solution, it became clear that the ELK stack was key to unifying each solution and providing a way for administrators to use each solution in a simple and accessible way.

Conclusions

Ultimately, Trusted Research Environments are still a very young field and there is much more work to be done, both from a security perspective and in terms of general functionality. The field is rapidly changing, and over time as it matures, more concrete ideas of what TRE infrastructure should ideally look like will emerge.

The Gitlab code ingress solution has proven to be a real possible option for TREs to integrate into their infrastructure. This work has demonstrated that it would allow administrators to build controls over what is allowed and easily integrate them with the CI hooks. As demonstrated in the POC, it can be done with a very small amount of code. More work needs to be done when it comes to detecting bad code automatically.

When it comes to egress, it's important to consider just how vital of a process this is. The reason egress controls are currently manual is because of the risks posed by what could happen if data could be exfiltrated. However, there are times when manual checks will fail, and this research shows that parts of the process can be automated to assist with the manual checks.

The ELK stack has proven to be a comprehensive solution for creating audit trails of user activity. The POC implementation here only demonstrates a small part of its potential, and for environments that don't currently have any activity monitoring it would greatly improve their security.

It's important to note that in the real world, all of these solutions would require people to develop, integrate and maintain them. This is especially true of the ELK stack, where truly comprehensive event monitoring would require a team of people looking for events and developing new ways to detect anomalous ones. Ultimately, this may not be feasible for a lot of institutions.

There are many more unsolved problems when it comes to TREs, and all of the solutions here have their limitations. However, this project has provided many suggestions to the key problems that TREs are currently facing, and successfully demonstrated how they might be implemented and integrated into existing infrastructure.

7.1 Future Work

There are some ideas presented within this paper that could have been developed further, and some ideas that were excluded from the paper for the sake of brevity. This section will mention some of those and discuss how they might be explored.

Identifying bad code is a problem that is a fundamental part of code ingress, since it's very difficult to determine what is and isn't "malicious" code. There could be entire dissertations just on this topic alone, but some simple heuristics could be determined and used to perform at least basic checks.

For example, if the code makes use of any libraries that involve networking, that doesn't make sense for within a TRE since they're isolated from the internet. Or if the code uses libraries for encoding or encrypting data, that should be a red flag since the data coming out should all be visible to the administrators.

As discussed in the design chapter, making use of a local docker registry would allow users to pull in commonly used containers in order to easily deploy applications they might want to make use of, without having to connect to the internet. For this project, it wasn't deemed as interesting of a component to implement as the other three, and was therefore only described conceptually.

It would still be interesting however to deploy this within the context of the existing proof-of-concept infrastructure. Particularly, it could be useful to create logging integration and record interactions with the registry to the Elasticsearch database to be visualised with Kibana.

One part of this project that would have been implemented given more time was communication from the egress container to the ELK stack, so that the administrators could view everything through the single unified interface of Kibana. There are parts of the egress code where it's indicated that the code would "flag for review", and these would send logs to Elasticsearch that could be displayed on a custom Kibana dashboard, or even trigger alerts that could send emails to the admins.

Ultimately, the future work to be done in this field is vast. It was important to limit the scope of this project to make it feasible to complete within the given time frame, but there are many more interesting problems and ideas that could be explored in the future.

References

- Alexa (2020). *Managing compute environments for researchers with Service Workbench on AWS - AWS Feed*. Section: Open Source. URL: <https://awsfeed.com/whats-new/open-source/managing-compute-environments-for-researchers-with-service-workbench-on-aws>, <https://aws.amazon.com/blogs/opensource/managing-compute-environments-for-researchers-with-service-workbench-on-aws/> (visited on 30/03/2021).
- Article 29 Data Protection Working Party (10th Apr. 2014). ‘Opinion 05/2014 on Anonymisation Techniques’. In: URL: https://ec.europa.eu/justice/article-29/documentation/opinion-recommendation/files/2014/wp216_en.pdf (visited on 08/03/2021).
- El Emam, K. and C. Alvarez (1st Feb. 2015). ‘A critical appraisal of the Article 29 Working Party Opinion 05/2014 on data anonymization techniques’. In: *International Data Privacy Law* 5.1, pp. 73–87. ISSN: 2044-3994, 2044-4001. DOI: 10.1093/idpl/ipu033. URL: <https://academic.oup.com/idpl/article-lookup/doi/10.1093/idpl/ipu033> (visited on 08/03/2021).
- El Emam, K., S. Rodgers and B. Malin (20th Mar. 2015). ‘Anonymising and sharing individual patient data’. In: *BMJ* 350 (mar20 1), h1139–h1139. ISSN: 1756-1833. DOI: 10.1136/bmj.h1139. URL: <https://www.bmj.com/lookup/doi/10.1136/bmj.h1139> (visited on 08/03/2021).
- Elastic (2021). *Winlogbeat: Analyze Windows Event Logs*. Elastic. URL: <https://www.elastic.co//beats/winlogbeat> (visited on 27/04/2021).
- Google (2021). *Google Trends*. Google Trends. URL: <https://trends.google.com/trends/explore?date=2009-11-01%202021-05-03&q=splunk,%22elk%20stack%22%20%2B%20%22logstash%22%20%2B%20%22elasticsearch%22%20%2B%20%22kibana%22> (visited on 03/05/2021).
- Jones, Kerina H. et al. (2019). ‘A Profile of the SAIL Databank on the UK Secure Research Platform’. In: *International Journal of Population Data Science* 4.2. Number: 2. ISSN: 2399-4908. DOI: 10.23889/ijpds.v4i2.1134. URL: <https://ijpds.org/article/view/1134> (visited on 11/03/2021).
- Kessler, Gary (2nd Mar. 2021). *File Signatures*. GCK’S FILE SIGNATURES TABLE. URL: https://www.garykessler.net/library/file_sigs.html (visited on 01/05/2021).

- Lea, Nathan Christopher et al. (21st June 2016). ‘Data Safe Havens and Trust: Toward a Common Understanding of Trusted Research Platforms for Governing Secure and Ethical Health Research’. In: *JMIR Medical Informatics* 4.2. Company: JMIR Medical Informatics Distributor: JMIR Medical Informatics Institution: JMIR Medical Informatics Label: JMIR Medical Informatics Publisher: JMIR Publications Inc., Toronto, Canada, e5571. DOI: 10.2196/medinform.5571. URL: <https://medinform.jmir.org/2016/2/e22> (visited on 07/03/2021).
- Mackay, Daniel F. et al. (6th Mar. 2012). ‘Impact of Scotland’s Smoke-Free Legislation on Pregnancy Complications: Retrospective Cohort Study’. In: *PLoS Medicine* 9.3. ISSN: 1549-1277. DOI: 10.1371/journal.pmed.1001175. URL: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3295815/> (visited on 30/03/2021).
- Masek, Pavel et al. (18th May 2018). ‘Unleashing Full Potential of Ansible Framework: University Labs Administration’. In: Proceedings of the XXth Conference of Open Innovations Association FRUCT. Vol. 426. DOI: 10.23919/FRUCT.2018.8468270. *Proxmox* (2021). URL: <https://www.proxmox.com/en/> (visited on 24/04/2021).
- Russinovich, Mark (2021). *Sysmon - Windows Sysinternals*. URL: <https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon> (visited on 27/04/2021).
- Shibani, Moza Al and Anupriya E (31st Oct. 2019). ‘Automated Threat Hunting Using ELK Stack - A Case Study’. In: *Indian Journal of Computer Science and Engineering* 10.5, pp. 118–127. ISSN: 22313850, 09765166. DOI: 10.21817/indjcse/2019/v10i5/191005008. URL: <http://www.ijcse.com/abstract.html?file=19-10-05-008> (visited on 02/05/2021).
- Smeeth, Liam et al. (11th Sept. 2004). ‘MMR vaccination and pervasive developmental disorders: a case-control study’. In: *The Lancet* 364.9438. Publisher: Elsevier, pp. 963–969. ISSN: 0140-6736, 1474-547X. DOI: 10.1016/S0140-6736(04)17020-7. URL: [https://www.thelancet.com/journals/lancet/article/PIIS0140-6736\(04\)17020-7/abstract](https://www.thelancet.com/journals/lancet/article/PIIS0140-6736(04)17020-7/abstract) (visited on 30/03/2021).
- Son, Sung Jun and Youngmi Kwon (Nov. 2017). ‘Performance of ELK stack and commercial system in security log analysis’. In: *2017 IEEE 13th Malaysia International Conference on Communications (MICC)*. 2017 IEEE 13th Malaysia International Conference on Communications (MICC), pp. 187–190. DOI: 10.1109/MICC.2017.8311756.
- Trusted-AI (4th May 2021). *Trusted-AI/adversarial-robustness-toolbox*. original-date: 2018-03-15T14:40:43Z. URL: <https://github.com/Trusted-AI/adversarial-robustness-toolbox> (visited on 04/05/2021).

Chapter A

Appendices

A.1 Git Diode Ingress Script

```
#!/bin/bash

function check_code() {
    if grep -rq "bad" $1; then
        return 1
    else
        return 0
    fi
}

# check for project name argument
if [ $# -eq 0 ]; then
    echo "No arguments supplied"
    exit 1
fi

# scan code
if check_code .; then
    echo 'code is fine'
    cp -r . /mnt/ingress/$1/
    echo 'code tranferred to ingress share'
else
    echo 'code is NOT fine'
    # flags for inspection
    echo 'code not transferred, flagged for review'
fi
```

A.2 Kibana Dashboard



Figure A.1: Kibana dashboard with three lenses

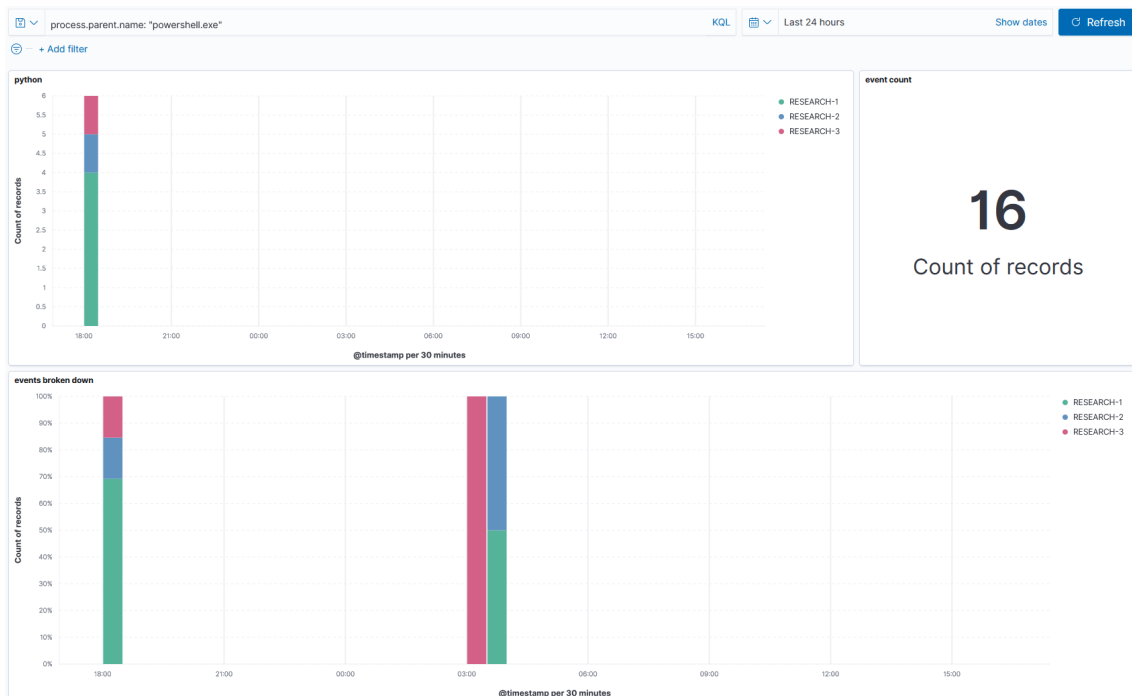


Figure A.2: Kibana dashboard filtering for child processes of powershell

A.3 Egress Scripts

A.3.1 scan bash script

```
#!/bin/bash

scan_path="/mnt/egress/research-1/"
cache_file="./size_cache"

if ! $(du -ab $(find -L $scan_path) | cmp -s "$cache_file"); then
    echo "changes detected"
    /usr/local/bin/python3.9 /root/type_check/type_check.py $scan_path
    du -ab $(find -L $scan_path) > "$cache_file"
else
    echo "no changes detected"
fi
```

A.3.2 type_check Python script

```
import os
import sys
import json

with open(os.path.join(os.path.dirname(os.path.abspath(__file__)),
    "types.json")) as f:
    types = json.loads(f.read())

def check(b, ext):
    stream = b.hex(' ')
    detected = False

    for t in types:
        for signature in t['signature']:
            offset = t['offset'] * 2 + t['offset']
            if signature == stream[offset:len(signature) + offset].upper():
                # file type has been found
                detected = True
                print('file type is', t['type'])
                if (t['extension'] != ext[1][1:]):
                    print('extension mismatch detected!')

    if not detected:
        print('no file type detected, probably plain text')

if __name__ == '__main__':

    for root, dirs, files in os.walk(sys.argv[1]):
        for name in files:
            print(os.path.join(root, name))
            with open(os.path.join(root, name), 'rb') as f:
                check(f.read(128), os.path.splitext(name))
```