University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Designing an Automated Cybersecurity Threat Mitigation System

Bayer Radiology

Mississauga, Ontario, Canada

Prepared by
Arthur Chun-Yin Leung
ID 20601312
userid ac7leung
2B Computer Engineering
23 November 2018

200 University Ave W
Waterloo, Ontario, Canada
N2L 3G1

23 November 2018

Vincent Gaudet, Chair
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Sir,

This report, entitled ‘‘Designing an Automated Cybersecurity Threat Mitigation System’’ was prepared as my 2B Work Report for the University of Waterloo. This report is in fulfillment of the course WKRPT 301. The purpose of this report is to document the process on selecting technologies to be used in creating an automated system for performing upgrades, testing, and reporting defects for virtualized computer systems.

Bayer Radiology specializes in developing software to interface with medical imaging devices, for both private and government healthcare entities. The Automation department of the Radimetrics team that I worked in was headed by Alex Bi, who oversaw the testing and deployments of the web application. We also worked closely with the QA department which defines various protocols for testing the Radimetrics web application.

I’d like to thank Alex Bi on providing guidance on the design process of the project outlined on this report. I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Arthur Chun-Yin Leung
ID 20601312

Contributions

For the duration of this co-op term, I worked in Bayer Radiology's Radimetrics team, which consisted of around 30 members. The team was mainly organized into the ''2.X'' (Existing product) and ''3.X'' (New product currently under active development), and was further sub-categorized by role such as Development, Testing, or Customer Support.

Radimetrics team's main goal was to develop a web based application for viewing, retrieving, and storing medical examinations. These examinations or studies would contain images from CT and Xray scanners, accompanied by the dosage information contained in a report. A main problem in modern healthcare systems is to track each patient's radiation dosage, to ensure they have not exceeded the safe amount prescribed over a fixed time period; Radimetrics provides physicians with software tools in order to address this problem. This web application was originally implemented in Adobe Flex, which has reached its end-of-life support at the time of writing according to Adobe's official source. While plans to use a modern Javascript framework (React) for the 3.X release were in place, maintenance and testing of previous releases were still in order. Tasks for each member of the team ranged from researching suitable technologies and implementations to bug fixing, testing, and provisioning servers for existing customers.

My primary role in the team was to research, develop, and maintain an automated solution to test and report any bugs in the existing 2.X application, and ensure that upgrades to the application server's operating system would not cause any unexpected behaviour. Since the transition to a JavaScript based application would happen eventually, compatibility was also a major consideration in designing this automated testing solution. My daily tasks ranged from implementing test protocols in code to researching and documentation of development related processes. Responsibilities in development operations (DevOps) such as maintenance of the network and provisioning of virtual machines used by the development and testing teams were also delegated to me. As such, though my title was ''Software Development/Tester'', I took on various tasks from other roles and learned in the process while gaining a broad skillset.

Clients of the Radimetrics team include hospitals and clinics in both private and public sectors and most of these clients use modern Enterprise Linux servers (RedHat, CentOS)

as part of their information systems. The Radimetrics product could then be delivered to the customer as an installer executable, or as a bare metal virtual machine image; an entire operating system that is run by a hypervisor host. These projects would require a team member to travel to a remote site per the customer's requirements, and provision the real server in person. Such one-off configurations at customer sites were common for clients who are government entities, as they have stringent security standards; often in the form of a checklist. Prior to shipping the Radimetrics product as a virtual machine image, it must undergo a ''hardening process'': Software updates and patches to eliminate vulnerabilities would be applied, with the goal to mitigate all attack vectors. This process also involves documenting each change meticulously, followed immediately by a round of thorough testing of the application and operating system. This was a tedious process done manually, and there existed no automated solution to accomplish this prior to my arrival. Thus one of my initial responsibilities within the Radimetrics team also included the manual testing of said virtual machines for clients, before I laid the groundwork to realize an automated solution later.

This report was written to describe an implementation of such an automated solution to the given problem during my co-op term at Bayer Radiology, and to outline the process of selecting software alternatives in realizing the solution. Though my work less pertains to the core application development, and more to the testing of the web application itself, the approach I took to design new test cases would most closely resemble that of ''black box'' testing. Thus, the efforts exerted in implementing the solution and preparing this report has collectively advanced my understanding of the DevOps process.

The effort I have invested in preparing this report has also furthered my communication and analytical skills; skills of which I believe are essential to developing professionalism, and are indispensable in all engineering workplace and academic settings.

In the broader scheme of things, the automated tests I have implemented will continue to serve its purpose. The solution can be scaled to accommodate the testing of additional software releases, and also serve as a guide to automate other manual processes.

Summary

The scope of this report is to document the implementation of an automated solution for testing of a web application software, with considerations such as compatibility and re-usability as the software approaches its end of life (and a new software release to succeed it).

The major points covered in this report are the requirement and criteria selection of an ideal automated solution, the importance of each criteria in this scenario, and comparison of design alternatives. A recommended solution optimized for the problem described would then be constructed from these available design options.

The major conclusions in this report are that testing solutions should be designed with scalability in mind, and how cloud computing can help realize such a solution. However, automation should not be treated as a solution to all problems, as the costs may not be justified in all cases. When used appropriately though, it could complement the productivity of manual QA testers, and benefit the development team as a whole. If the team already does have an automated solution, spending time to optimize and refactor the codebase could be beneficial in terms of overall performance.

The major recommendations in this report are that development teams should consider commercial technologies even if open source options are more popular. Features and benefits that these options provide could greatly reduce the efforts in implementing the solution, and in turn can reduce time spent maintaining said solution. It is also highly recommended that solutions for documentation and notification for the development process be explored, to prevent bugs being left unnoticed or working knowledge being lost among team members.

Table of Contents

List of Figures

List of Tables

1 Introduction

This section details the background information and motivation leading up to the design of this automated solution. In addition, the expected scope of the project will also be outlined, as a preface to the nature of the problem.

1.1 Background

The concern of data breaches due to poor security implementations are ever-growing, especially in light of recent cases. As a healthcare corporation, Bayer Radiology takes all appropriate measures to guard Personally Identifiable Information (PII) of patients, such that any damages would be mitigated in the event of a breach. Measures taken also include precautionary ones; for instance, keeping the operating systems of all machines up to date in order to close any vulnerabilities, and hence minimize the number of attack vectors that may be exploited.

When performing security upgrades however, compatibility is a critical aspect. A new and untested software update to an operating system could break other features that were previously functional. For this reason, the Radimetrics team utilizes virtual machines (VMs) on a cloud hosting service for the purpose of testing on different operating system versions. Each virtual machine can be running either CentOS or RedHat Enterprise Linux (RHEL) which are common operating systems for enterprise application servers[1]. This VM network also simulated a real world hospital network, where patients' information and imaging data could be sent between each machine or ''node''.

There were two independent networks of hypervisor hosts: ''VHN'' and ''HP Gen'' refer to ''Virtual Hospital Network'' and the server lineup from the manufacturer Dell respectively (See Figure 1). VMs from both networks are accessed through reverse SSH tunnelling via a central ''support server'' (See Figure 2) for manual testing purposes. This network was designed with scalability being the main consideration; adding a new virtual machine would only require mapping a few additional ports.

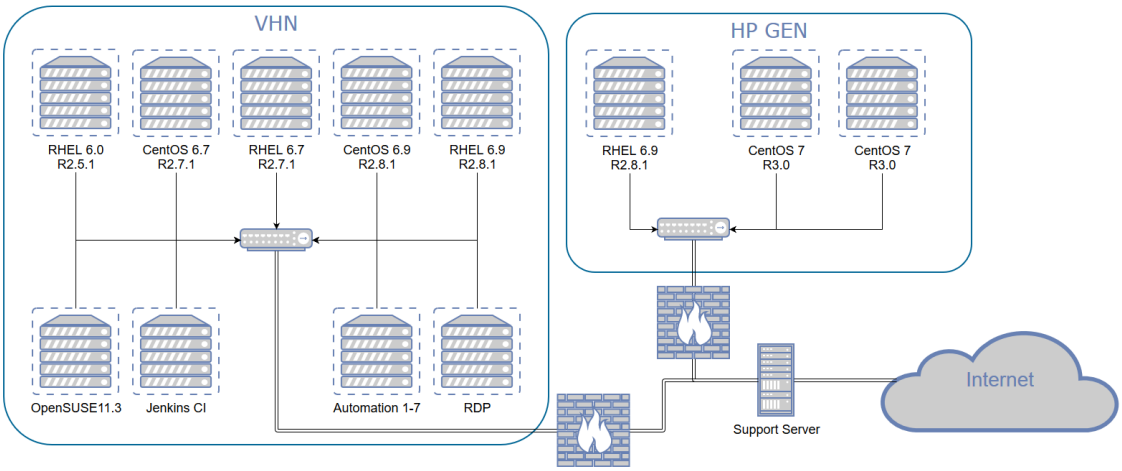

Figure 1. An overview of the development and testing network topology

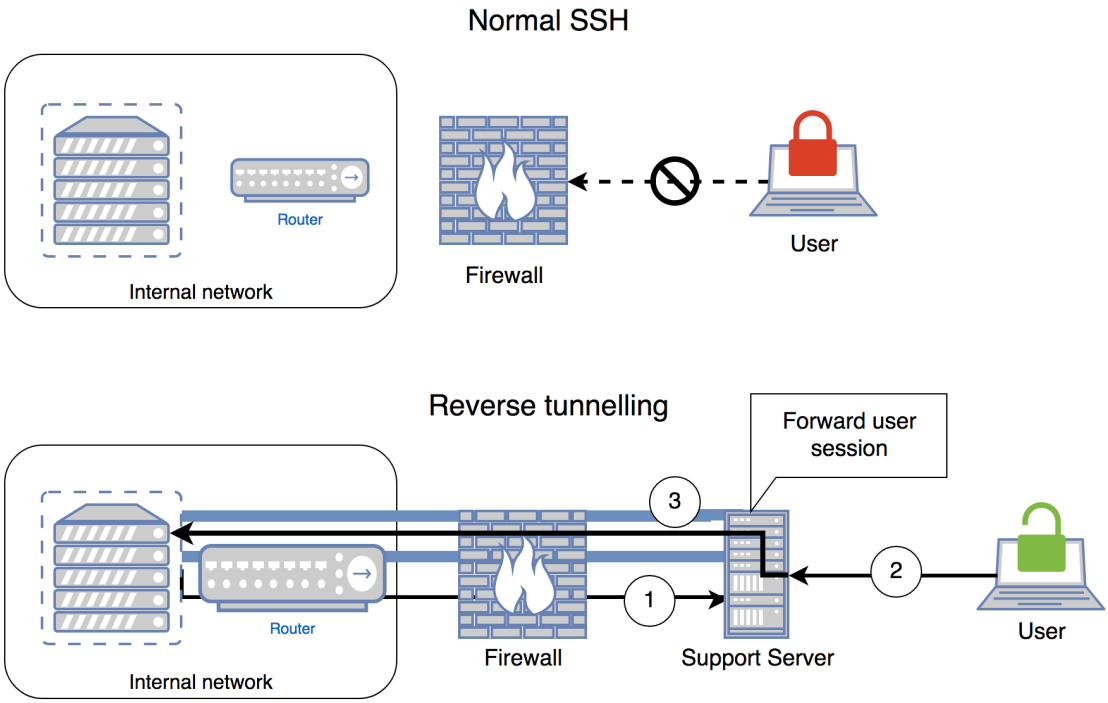

Figure 2. Illustration of how reverse SSH allows inbound connections through a firewalled network

1.2 Motivation

With up to dozens of such virtual machines deployed simultaneously for development and testing purposes, it can become an arduous task to manually ensure that each machine has the

latest security updates. The fact that these machines may have different operating systems, with different releases of the web application running further complicates matters. This calls for an automated solution, where ideally the upgrades are performed automatically, then a round of testing is performed to ensure that the functionalities of the Radimetrics application remains unaffected.

1.3 Scope and considerations

This report is mainly concerned with strategies and methods to improve the efficiency of the QA and testing team. Any software development project that is cloud based and utilizing multiple virtual machines can likely implement a similar solution, and see the benefits outlined in the following report compared to alternatives also mentioned below.

At the time of writing, the solution only had to address 3 major releases of the Radimetrics web application, across 2 operating systems; a total of 5 combinations as listed in Table 1

OS	OS Version	Radimetrics Release
RHEL	6.3	2.5.1b
CentOS	6.7	2.7.1
RHEL	6.7	2.7.1
CentOS	6.9	2.8.1b
RHEL	6.9	2.8.1b

Table 1. 5 different combinations of the Radimetrics application and Operating System to be tested

The application itself is built with Adobe Flex. Though now a legacy piece of software no longer under active development, maintaining support for existing customers are a priority in addition to provisioning upgrades. Prior to this solution, manually testing every functionality in the application was achieved by following a protocol. There already existed a set of ‘‘System Tests’’ which were designed to run on the backend and test certain features, including optical character recognition (OCR) and garbage collection mechanisms of the software. These tests were designed to verify the functionality of the backend and integrity of data, and are designed to be unit tests. The need for a unified automated solution became evident, and the following sections will cover the process from requirements through analysis, to reach a practical implementation.

2 Requirements

In order to construct a solution, a good understanding of the problem should be attained. Once that is done, definition of the requirements is the next step, and after that the selection of criteria to judge the effectiveness of the solution. The scope and considerations as previously discussed are the major factors during this selection process.

2.1 Summary and Criteria selection

The main goal was to build an automated system to upgrade each virtual machine's operating system, while testing for any incompatibilities with the application due to these upgrades. The solution was to be automated as a Jenkins Continuous Integration pipeline, with a means of reporting any uncovered defects to the development team. Part of the requirement was also to design the tests to be as reusable as possible, as the Radimetrics team had plans to migrate to a JavaScript frontend in the near future.

The requirements, or characteristics of the ideal solution are as follows:

- Automated, with no human involvement other than the start and end of testing
- Modular; easily upgradeable and maintainable
- Robust; failure handling capabilities a single test failing should not affect other tests
- Compatible with existing means of automation (such as Jenkins)

From these requirements we select the following criteria:

- Time cost: How much time is required to fully realize the solution? (Estimate in man hours, or equivalent metric)
- Financial cost: Are there licensing or support staff costs? Computational resources? Storage? (Estimate in dollars)
- Modularity: How decoupled are the test cases from each other? (Level of modularity/ customizability)
- Test Coverage: How comprehensive is the solution at testing all facets of the web application?

- Re-usability: Could this solution be used for newer iterations of the Radimetrics web application? (Level of re-usability)
- Feedback: Does this solution clearly describe any defects found during testing? (Feedback capability)

3 Design options

There may be many ways to implement a solution, but usually the goal is to do so with the most efficiency in terms of resources and time. As different options may offer attributes that suit certain requirements better than other alternatives, special consideration may be given to one option in certain circumstances. Generally however, quantitative analysis by means of decision matrices can be used to determine the suitability of each alternative; a solution can then be constructed using a combination of the best suited options found this way.

3.1 Summary of choices available for enhanced testing

To address the main goal of decreasing time wasted in manual/ repetitive testing, different technologies were considered based on the features that they offered, compatibility with the build pipeline, and upgradeability. Figure 1 illustrates the software system that was to be tested. It consisted of an Apache Tomcat application server, a Postgres relational database, and the main application written in Adobe Flex (Flash). From here on, both the database and application server will be referred to as the ‘‘backend’’, and the flash technology as ‘‘frontend’’ unless specified otherwise. This is illustrated by how the content is served from the browser (front facing) while the data processing and storage logic are abstracted behind the application server.

For the backend or serverside components of the web application, the choices were:

1. **Manual testing:** This option involves loading test data using solely the frontend, going through the process as if it were a customer/ hospital setting up the application for the first time. Data validation could be performed by querying the database directly.
2. **Open source and third party tools:** This makes use of open source tools or libraries

(DCMTK, Horos, HAPI TestPanel) or from commercial vendors (OsiriX), to facilitate the testing of sending and receiving medical imaging data and patient data.

3. **System (unit) testing:** A unit based approach for testing specific features; test data would be loaded as input and processed by the application logic. The final output would then be compared against a set of expected results, which determines whether the application is behaving normally.

For the frontend (UI of web application), the following were considered:

1. **Manual testing:** The application would be tested by a member in the QA team following instructions outlined in a formal test protocol.
2. **Ranorex:** This commercial UI testing technology is based on referencing all UI elements as XPaths, which then is stored in an object model repository. Automated tests can be built using this repository of UI elements with C# code[2].
3. **TestComplete:** This commercial UI testing technology is similar to Ranorex, which also provides UI automation and an object model repository. Automated tests can be built using this repository of UI elements with C# script code[2].
4. **Selenium (flash) WebDriver:** Since modern browsers such as Mozilla Firefox and Google Chrome provide WebDriver tools for developers, they can be used in automating web applications.

A weighing scheme was chosen according to the importance of each category in the considerations of this project, which is shown on Table 2. Each category of an option was then assigned a numerical score from 0 to 10 based on qualitative analysis, with 0 representing the worst and 10 being the best. For example, if a 10 is assigned to ‘‘Financial Cost’’ that would mean the option is the most cost-effective: this is the number to the left of the arrow ($\rightarrow$). The respective weight is applied to obtain the criterion score (ex. score of 5 with a 15% weight would yield 5 * 0.15 = 0.75) to the right of this arrow. The final score for each option was obtained by summing all the criteria scores across the row of the table.

Table 2. Criteria weighing

Criteria	T. cost.	F. cost.	Modu.	Cover.	Maint.	Re-use.	Feedback
Weight	20%	10%	15%	25%	5%	10%	15%

Table 3. Qualitative evaluation of backend testing options

	T. cost	F. cost	Modu.	Cover.	Maint.	Re-use.	Feedback.	**Score**
1. Manual testing	9 → **1.8**	9 → **0.9**	5 → **0.75**	5 → **1.25**	8 → **0.4**	6 → **0.6**	7 → **1.05**	**6.75**
2. 3rd party tools	5 → **1**	6 → **0.6**	4 → **0.6**	7 → **1.75**	6 → **0.3**	8 → **0.8**	8 → **1.2**	**6.25**
3. Unit testing	5 → **1**	7 → **0.7**	9 → **1.35**	10 → **2.5**	6 → **0.3**	5 → **0.5**	9 → **1.35**	**7.7**

3.2 Comparison of options for backend testing

The main purpose of testing the backend is to verify the integrity of data stored by the database and served by the webserver. Additional purposes include checking configurations such as file system structure and permissions, and prevent unauthorized access from other applications. Other tests could also be monitoring for memory leaks, software upgrade compatibility etc...

The first option of manual testing required little to no additional costs: a team member can be trained within a few days, and no other expenses would be incurred. However, it was given the lowest score in automation, as it would require the dedication of at least 1 tester for each instance of the application server. Moreover, it would be difficult to monitor all parameters of the operating system at once; such as in the case of a memory leak.

The second option of utilizing open source and third party tools scored well for time cost, maintenance effort, and feedback. Since these tools targeted specific functionality, they work out of the box and provide logging capabilities. Low scores were given for Coverage as the tools mainly tested the connectivity functionality, such as sending and retrieving of patient and imaging data. Although it is a major feature of the application, these tools cannot encompass every single feature that customers may need to use in the Radimetrics application.

The third option offered the best in automation and coverage, as a comprehensive unit test suite can incorporate different libraries for testing specific functions. Re-usability would suffer if the team ever decides to rewrite the entire backend of the system, but since it is not known at the time of writing, it is assumed to remain the same. This option would also result in the most time spent setting up and maintenance.

3.3 Comparison of options for frontend testing

The main purposes of testing the frontend were: Firstly, to ensure that user interface elements in the web application were being displayed in the correct location.

Secondly, to ensure that content served up to date and correct. Sometimes caching of old webpages could result in the application displaying of outdated data, and hence would not be desired. The application used SOAP APIs as the communication interface, but was not open to clients other than the web application developed by the Radimetrics team. From this we analyze the four options:

The first option of manual testing using a web browser received the worst score in Automation; there is none. However, a competent QA tester can learn after the initial trials, and provide better feedback with little maintenance and setup time in subsequent tests. The only drawback would be that WebDrivers usually require more memory and computational power, especially when being run for extended periods of time in automated testing; this point is revisited in Section 6.

The second option involves using the commercial Ranorex automation toolkit. It is a software automation solution that can emulate mouse and keyboard input for testing purposes. Ranorex operates by mapping all UI elements into a tree based on XPath, and saving each element into an object repository (Figure 3). Automation of the UI is done trivially simply by referring to each object, using a few lines of C# code (see Listing 1). In addition, automated image comparison is another feature offered that could be leveraged during testing. Licensing is based on a one time fee: each ‘‘premium’’ license for a developer costs $4,990 USD and additional ‘‘runtime’’ licenses for each automation server costs $890 USD each.

```
1 repository.SearchTextBox.Click();
2 repository.SearchTextBox.PressKeys("UWaterloo");
3 repository.SearchButton.Click();
```

Listing 1: Code snippet of how one would automate a search

The third option is to use TestComplete, another commercial tool like Ranorex. It uses

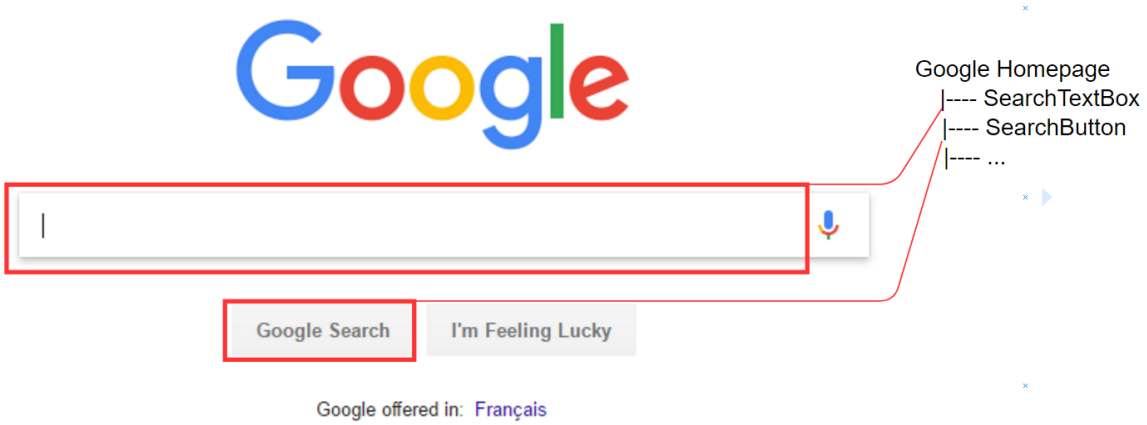

Figure 3. ‘‘Tree’’ structure of UI elements: search box and button are separate elements but belong to the same parent web page

Table 4. Qualitative evaluation of frontend testing options

	T. cost	F. cost	Modu.	Cover.	Maint.	Re-use.	Feedback.	**Score**
1. Manual	3 → **0.6**	9 → **0.9**	5 → **0.75**	6 → **1.5**	9 → **0.45**	6 → **0.6**	5 → **0.75**	**5.55**
2. Ranorex	10 → **2**	3 → **0.3**	8 → **1.2**	9 → **2.25**	6 → **0.3**	8 → **0.8**	8 → **1.2**	**8.05**
3. TestComplete	8 → **1.6**	4 → **0.4**	8 → **1.2**	9 → **2.25**	6 → **0.3**	5 → **0.5**	9 → **1.35**	**7.6**
4. Selenium	5 → **1**	10 → **1**	7 → **1.05**	8 → **2**	5 → **0.25**	4 → **0.4**	6 → **0.9**	**6.6**

C#Script for building automation flows, and can automate desktop, web, and mobile technologies. Licensing model is also similar to Ranorex’s: each developer or platform ‘‘bundle’’ costs $7,000 USD and additional automation agent costs $699 USD. Additional modules must be purchased separately unlike Ranorex, which includes all features such as mobile testing under a premium license.

The fourth option of using Selenium webdriver to test offered good automation capabilities. Being an open source technology it is free, albeit less developer-friendly compared to the features that commercial automation tools offer. As such it received lower scores for setup and time cost. Its report generation capabilities depend on which library is used, but is generally similar if not better than the commercial tools previously mentioned. In addition, cross browser testing must be implemented manually (separate webdrivers for Internet Explorer and Firefox, for example). However, one of its main advantage is its popularity (over 30% market share)[3], so adoption is widespread and usually solutions exist to any problems encountered during testing.

3.4 Possible solutions

After performing the qualitative analysis of the technologies available for automation purposes, the following viable solutions that use a combination of frontend and backend tools were proposed:

1. **Manual testing of frontend with third party tools for backend:**
   This solution would involve the most human action, as the tools serve to reduce the time taken rather than completely automate the process. It requires no additional computing resources apart from the dedication of a QA team and their workstation. Sample imaging and patient data could be preloaded on a shared DICOM node and fetched at the time of testing, and sending of new patient examinations can be done through user friendly utilities such as OsiriX or Horos.
2. **Ranorex automated testing with Unit testing backend:**
   This solution makes the most use of automation, but is the most costly due to licensing and computational resources. Depending on the implantation, the system may be scaled horizontally[4] by executing test agents in parallel, thus actual runtime can be made inversely proportional to number of agents available assuming that load is balanced equally:
   $$ActualRuntime = \frac{TotalRuntime}{NumberOfAgents}$$
   Where total runtime is the time taken if a single agent were to run every test case in the suite sequentially.
   A single Jenkins job can be scheduled to run this solution however often is desired, but also introduces a responsibility to update and maintain unit tests.
3. **Selenium WebDriver frontend tests with third party tools for backend:**
   The web application will be tested using Selenium webdriver tool on the desktop. This should be able to cover most frontend use cases of the web application. Backend testing would be achieved with Selenium flash which uses Java code (and likely Maven for dependency management). As both Java and Selenium are free technologies, this solution would be most appealing from a financial cost standpoint, as it only requires the purchase of additional computational resources while offering a similar level of automation. However, it may take more time to setup and maintain compared to using

commercial options.

4. **TestComplete frontend with Unit testing backend:**
   Using TestComplete instead of Ranorex would mean a cheaper horizontal scaling cost (considering the cost of a single runtime license). However, that would mean giving up IDE support (such as Visual Studio) when writing test cases, and more time spent learning a third party scripting language. Automation capabilities would still be considered better than any manual alternative, and it offers Selenium support as well.

4 Analysis of proposed solution

A high level overview of each step in the Jenkins pipeline is illustrated in Figure 4

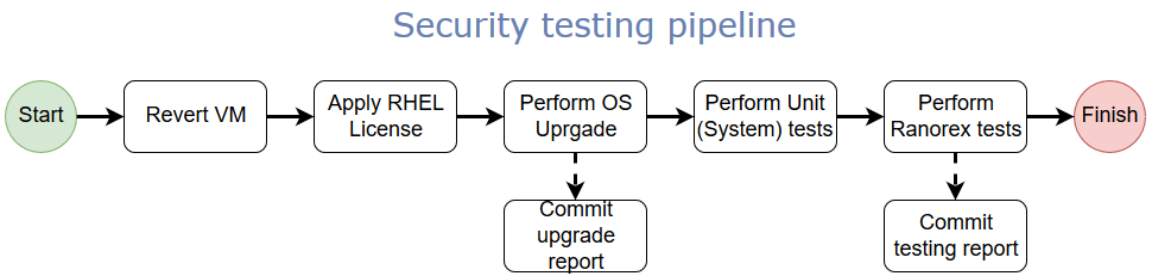

Figure 4. Sequence of jobs within the automated solution pipeline

Ultimately, it was determined that Ranorex should used over other commercial tools such as TestComplete. Although the latter had a less costly scaling option, it would require more time to learn a proprietary scripting language; whereas a developer is more likely to already have exposure to the popular Visual Studio to begin development with the Ranorex framework. Also as C# is a fully featured object oriented language supported by Microsoft, functionality of .NET libraries by other developers can also be incorporated when writing tests, which makes it more powerful than a scripting language by itself. Lastly, Ranorex includes a ‘‘guaranteed initial response time of 24 hours or less’’[2] in their enterprise customer policy, which can help minimize downtime of testing, and in turn lower the response time to Radimetrics’ customers.

This solution demonstrates good modularity using a customizable ‘‘runner’’ file, where individual test cases could be selected to run. This meant that the level of coverage could be customized on demand. For example, one may wish to perform ‘‘smoke testing’’, where

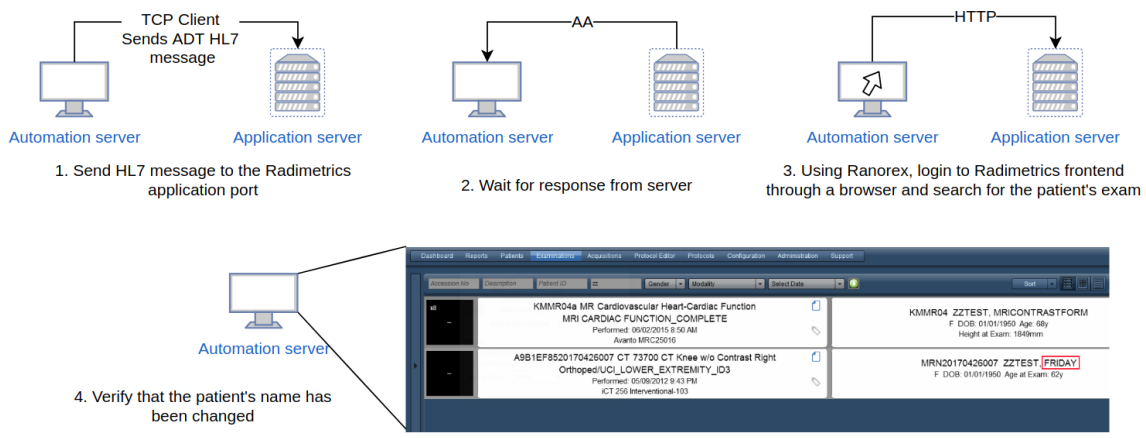

Figure 5. Example test case for testing HL7 inbound functionality

only the essential functions of the application are checked. In the case of Radimetrics' web application, the main functionalities included sending and receiving study data. Most of this communication occurs over TCP channels, and thus can be automated with C# code (See Appendix A).

An example test case could be verifying the inbound functionality of the application: it should be able to receive HL7 messages from other applications, for administering patients in a hospital.

The following sequence of steps can be fully automated as illustrated in Figure 5:

As demonstrated, these Ranorex tests generally follow the ‘‘black box’’ testing methodology; the internal state of the application is not tested, but given certain inputs, the system's behaviour or output should correspond to expected results.

In the end, this solution was estimated to take 1 month to implement, and in reality took little over 5 weeks for I was the only developer of this solution. Due to limitations of the automation toolkit, certain test cases could not be realized fully and required some workarounds. This included the switching of active directory users on Windows based systems to test a Single Sign On feature, but was determined that manual testing of this feature can be done.

Costs were mainly attributed to the cloud hosting of these VMs. According to the rackspace hosting service used by the Radimetrics team, a single general purpose Linux server instance

is around $108 per month[5]. Additional servers added would incur this scaling cost, as this solution expands to accommodate later software releases (Radimetrics 2.9 or 2.10).

4.1 Setup

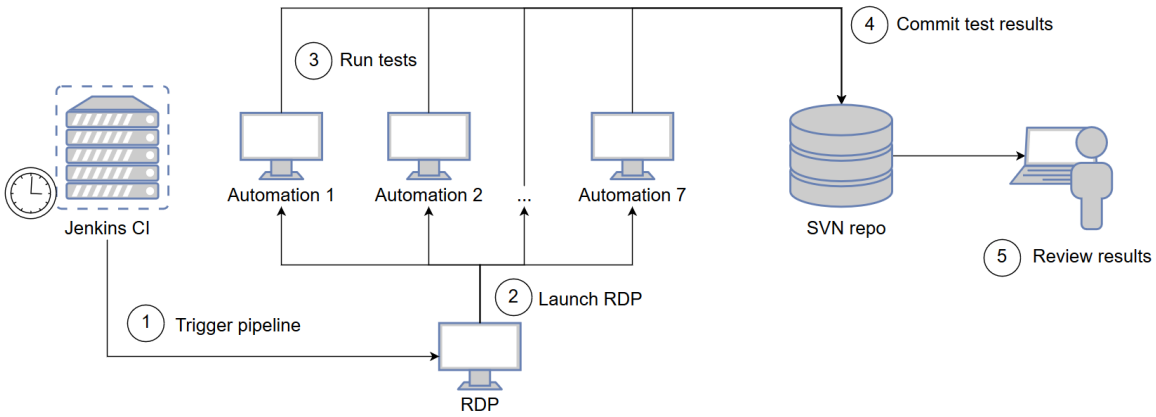

Figure 6. Jenkins pipeline for frontend testing (Ranorex)

The proposed pipeline in Figure 6 requires the following sequence of jobs to run:

1. Revert VM to pre-upgrade conditions:
A snapshot of the VM was taken prior to upgrading, so reverting to a ‘‘clean’’ state before testing allows the process to be reproducible.
2. Applying RHEL License:
In order to install updates, a machine must be registered with an active RHEL subscription. As there were more VMs than subscription keys, they needed to be shared/ rotated each time a server is used for testing.
3. Perform OS upgrade:
This step runs the yum package manager [6] and performs a distribution upgrade, which updates the operating system to the latest minor release and applies all the necessary security updates. A log of all packages updated or installed in this process is also committed to the SVN[7] repository for logging purposes.
4. Perform Unit (System) tests:
Performs the testing of specific backend functionality (OCR, garbage collection, database schema etc...)
5. Perform Ranorex tests:

Performs the smoke test suite, which only tests the core functionalities of the application; basic funcitons such as logging in, sending data outbound and receiving inbound communications. Like in step 3, the reports produced by Ranorex would be committed to the SVN repository to be read later.

5 Performance

A total of 7 Windows test agents were configured to automate test cases. Servers 1 to 6 were reserved for running a full version of the automated test suite, which tested all possible use cases of the frontend. Server 7 was used to run a ‘‘smoke test’’ suite, where only the core and major connectivity features were tested.

Since the scope of this project was to test the major features following an operating system upgrade, it was determined that a ‘‘smoke test’’ would suffice instead of running the full suite every time. This smoke test took around 2 hours compared to 8 hours if the full test suite were to be run instead. In addition, CPU time can be saved as the smoke test only required 1 automation server to be powered on (instead of 6).

The implementation of this solution successfully fulfilled all the requirements described in section 2.1 with the following characteristics:

Automation: The Jenkins pipeline can be set up to run daily, and configured to rotate through all permutations of Operating System and Radimetrics versions. At the time of writing there were 5 servers, so it can be set up to rotate through one instance per weekday.

Modularity: By specifying boolean parameters in the Jenkins pipeline, certain steps may be skipped. For instance, setting the flag to run ‘‘System Test’’ option to ‘‘false’’ skips the running of unit tests in the pipeline. As for the VM network, a new VM can be added any time to the ‘‘VHN’’ network for testing, as it would be in the same subnet as the Jenkins CI and frontend automation servers. If a new VM is added to the ‘‘HP Gen’’ network, a simple portforwarding entry would need to be added to the network router, which should take no more than a few minutes. These characteristics prove the solution to be a flexible and modular one.

Robustness: Each time the test is run, the VM is reverted to a ‘‘clean snapshot’’. This ensures that changes made from previous runs of the test do not affect the current test’s results. As mentioned in the above point as well, individual test cases may be skipped without changing the behaviour of other cases. The backend test suite was also scheduled daily, but could also be run as needed since it took 5 to 20 minutes per test case.

Feedback: After the operating system is upgraded, a logfile was created detailing all new software that was installed or upgraded, then committed to the SVN repository. Then after the smoke testing finishes, reports of each test case are also committed to SVN where the developers and testers can later read. The automated reports contain detailed logs and screenshots. Video capture of the entire automated process is another helpful feature that can help identify defects, as well as document how to reproduce them.

6 Additional optimizations and improvements

In order to improve the scalability and re-usability of this automated test framework, it would be preferable to automate the load balancing of test cases so load can be distributed evenly across all automation servers. This would be a generalized ‘‘k-partition’’ problem [8], where the goal is to minimize the difference between each partition’s sum. To illustrate with a simple example, suppose there are 3 servers which can each execute a test case at a time, independently of each other. There are 5 test cases, which take 10, 25, 40, 50, and 60 minutes respectively. The optimal solution is illustrated in Figure 7 which minimizes the difference to 5 minutes; so all test cases will be completed within 65 minutes. A solution is possible using a dynamic programming greedy algorithm [9], but further discussions would fall outside the scope of this report.

7 Conclusions

It was concluded that using commercial automation software was the best option to realize this solution; it was determined that the features it offered outweighed the financial costs, as discussed in the qualitative analysis and performance sections. The modularity of test cases in Ranorex allowed levels of testing to be customized to the testing and development

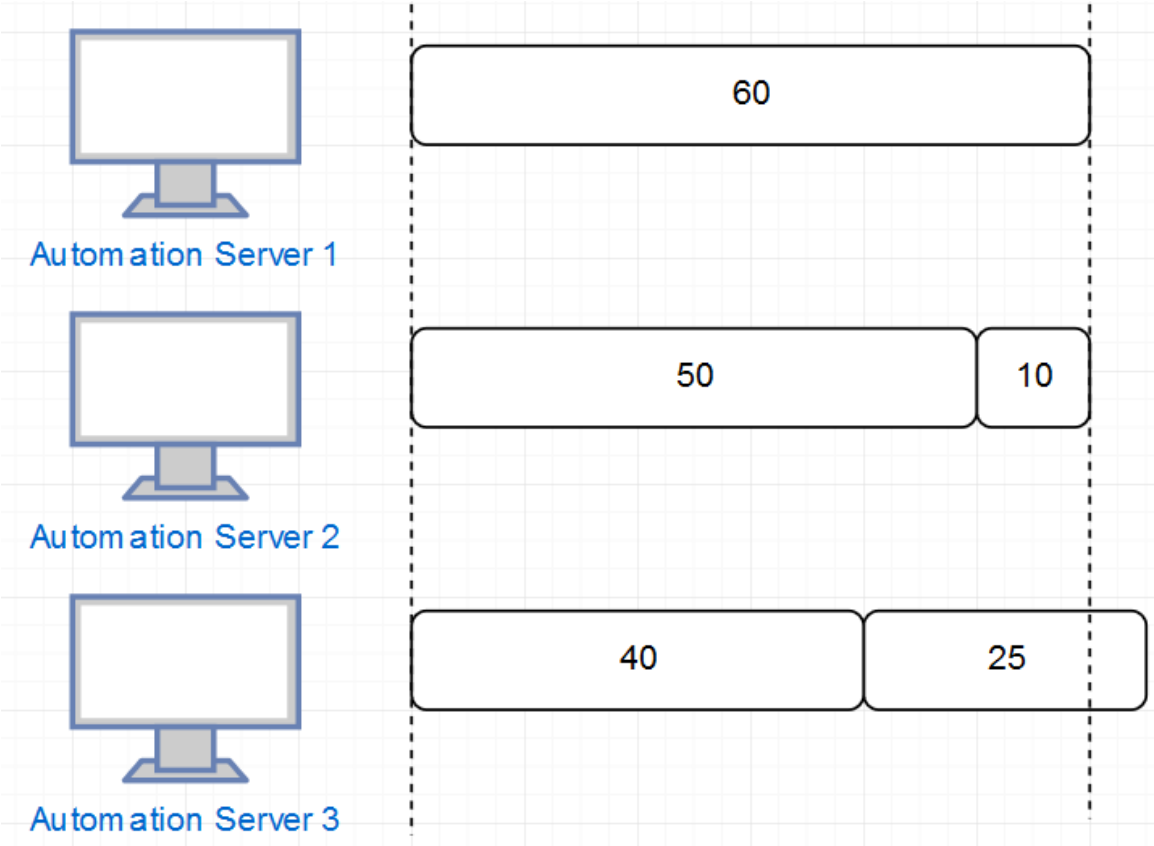

Figure 7. The optimal solution with k = 3 and 5 elements to be packed

team's needs, which was one of the major criteria. A high level of automation was also achieved with a Jenkins pipeline, to prepare the VMs before testing (reverting snapshots, applying licenses) as well as reporting the results at the end of testing. Overall, it met all the requirements set initially, as a preventative measure to address cybersecurity concerns.

8 Recommendations

Based on the analysis and conclusions in this report, it is recommended that the developers of the new ‘‘3.X’’ Radimetrics software should consider a similar solution for automated testing. In addition, the following items should be considered in order to improve the existing testing solution:

- Research and implement an efficient load balancing algorithm to distribute test cases across frontend testing servers

- Research a more robust method to test communication between frontend and backend. For the new 3.X product it is likely that JSON webservices will be utilized, therefore a suitable automated testing tool such as Postman[10] should be considered.
- Document best practices and guidelines for other developers and testers to use and expand the automated solution.
- Find a way to standardize the process of loading test data, so that this process can be automated as well in the future.

Glossary

ADT: Admit/ Discharge/ Transfer, a type of HL7 message that updates patients' information and status within software utilized by a hospital.

API: Application Program Interface, a set of defined subroutines of a software application that allows other components or applications to interact with the software..

DevOps: Development and Operations, practice that focuses on automating processes in software development, testing, and deployment/ release and make these processes more efficient.

DICOM: Digital Imaging and Communications in Medicine, an international standard for storing/ transmitting data between medical imaging devices.

HL7: Health Level Seven International, an international standard for storing/ transmitting data between medical imaging devices.

HTTP: HyperText Transfer Protocol, a protocol definition or rules for transferring and formatting of messages and data between information systems.

QA: Quality Assurance, a team or department in software development concerned with the testing of software, with the goal of delivering bug-free software to customers.

TCP: Transmission Control Protocol, a standard (RFC 793) by the Department of Defense that defines how programs communicate over a network in computer systems.

UI: User Interface, the means or components of software that the user will use to interact with the application, usually with visual or graphical elements.

VM: Virtual Machine, an emulated computer system that provides functionality of physical computers, such as running an operating system. Multiple virtual machines may be run by a single hypervisor or host machine.

WKRPT: Work-term report; the acronym used by the University of Waterloo Undergraduate Calendar.

XML: Extensible Markup Language, a human and machine readable format for encoding data, using enclosing tags.

XPath: Syntax for defining parts of an XML document, using path expressions to navigate the XML document. Expressions can be used to select nodes or node-sets in an XML document, such as web elements in an HTML document.

References

[1] *Comparison of the usage of Linux vs. Windows for websites*. Web Technology Surveys. URL: https://w3techs.com/technologies/comparison/os-linux,os-windows.

[2] *Ranorex*. Ranorex GmbH. URL: https://www.ranorex.com/.

[3] *Software Testing Tools products*. iDatalabs. URL: https://idatalabs.com/tech/software-testing-tools.

[4] David Beaumont. *How to explain vertical and horizontal scaling in the cloud*. Ed. by ibm.com. ibm.com [Online; posted 9-April-2014]. 2014.

[5] *Cloud Servers Pricing*. Rackspace Inc. URL: https://www.rackspace.com/cloud/servers/pricing#num3.

[6] Edgewall Software. *Yum Package Manager*. Version 3.4.3. May 4, 2018. URL: http://yum.baseurl.org/.

[7] *Apache Subversion Design Spec*. URL: http://svn.apache.org/repos/asf/subversion/trunk/notes/subversion-design.html#goals.

[8] Brian Hayes. ‘‘The Easiest Hard Problem’’. In: *American Scientist* 90.2 (2002), p. 113. DOI: 10.1511/2002.2.113.

[9] Utkarsh Trivedi. *Partition of a set into K subsets with equal sum*. URL: https://www.geeksforgeeks.org/partition-set-k-subsets-equal-sum/.

[10] *Postman*. Postdot Technologies, Inc., 2016.

Appendix A Example test case

HL7 ADT test case

The following snippet of C# code is used to send an ADT HL7 message. ADT stands for ‘‘Admit, Discharge, Transfer’’ and such messages are generally used to update patient information as well, such as given/ last names, date of birth etc.

The data in each field of the message is delimited with the pipe character — as shown in Listing 3 and the The identifier ‘‘ADT A08’’ is used to indicate the ‘‘Update patient information’’ event type.

The message is then encoded in a MLLP (Minimum Lower Layer Protocol) frame: it is prefixed with the ‘‘Vertical Tab’’ character and suffixed with a ‘‘File Separator’’ and ‘‘Carriage Return’’, before getting sent through the TCP connection.

```
public void SendHL7ADT() {
            string host = Config.IP;
            int port = Int32.Parse(Config.HL7Port);
            string[] adts = {
                "MSH|^~\\&|Vision Series RIS - PACS|RADIOLOGY IMAGING ASSOCIATES|PACS
    ||20161201140812||ADT^A08|1612011408125630|P|2.3.1\r\nPID|1|MRN20170426007|
    T818267||ZZTEST^"+dow+"||19500101|F|FORMS||^^^^^US||(303)111-1111|||U"
            };
            Report.Info("Day of week is " + dow);

            foreach (string adt in adts) {
                try {
                    // Encode the message into MLLP (Minimum Lower Layer Protocol)
    frame, then write to TCP stream
                    TcpClient client = new TcpClient();
                    Report.Info("Sending ADT: \n" + adt);
                    client.Connect(host, port);

                    ASCIIEncoding enc = new ASCIIEncoding();
                    byte[] b1 = { 0x0B };       // VT
                    byte[] b2 = { 0x1C, 0x0D }; // FS, CR

                    // MLLP frame:
                    // +----+-----------+----+----+
                    // | VT | <message> | FS | CR |
                    // +----+-----------+----+----+
                    List<byte[]> d = new List<byte[]>();
                    d.Add(b1);
                    d.Add(enc.GetBytes(adt));
                    d.Add(b2);

                    byte[] ba = d.SelectMany(a => a).ToArray();
                    Stream stm = client.GetStream();
                    stm.Write(ba, 0, ba.Length);

                    byte[] bb = new byte[1000];
                    int k = stm.Read(bb, 0, 1000);

                    string s = System.Text.Encoding.UTF8.GetString(bb, 0, k - 1);

                    client.Close();

                    Report.Info("Received reponse from server: " + s);
                    Report.Success("Sent HL7 message successfully");
                } catch (Exception ex) {
                    Report.Failure("Exception when writing to TCP: " + ex.Message);
```

```
                }
            }
        }
```

Listing 2: C# code to send an HL7 message

```
MSH|^~\&|Vision Series RIS - PACS|RADIOLOGY IMAGING ASSOCIATES|PACS||20161201140812||
    ADT^A08|1612011408125630|P|2.3.1
PID|1|MRN20170426007|T818267||ZZTEST^GIVENNAME|19500101|F|FORMS||^^^^^US||(303)
    111-1111|||U
```

Listing 3: Example of an ADT HL7 message to update the patient's name and other details