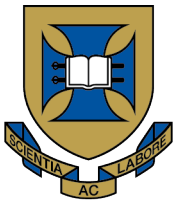

THE UNIVERSITY OF QUEENSLAND

AUSTRALIA

SCHOOL OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

THESIS

Automatic Patching of Security Vulnerabilities in Software

Conceptualising a UQ self defence system by using Patcherex to automatically insert code patches into vulnerable executables

Author:

Dospore

Supervisor:

Professor Ryan KO

Supported by:

Mr Taejun Choi and Mr Joshua Scarsbrook

22nd June, 2020

Mr Dospore

March 5, 2023

Prof Paul Strooper
Head of School
School of Information Technology and Electrical Engineering
The University of Queensland
St Lucia, Q 4072

Dear Professor Strooper,

In accordance with the requirements of the degree of Bachelor of Engineering in the division of Electrical Engineering, Electrical and Biomedical Engineering, Electrical and Computer Engineering, Software Engineering, Mechatronic Engineering, I present the following thesis entitled "Automatically Patching Security Vulnerabilities in Software Using Patcherex". This work was performed under the supervision of Prof. Ryan Ko, Mr Taejun Choi and Mr Joshua Scarsbrook.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,

Dospore.

Acknowledgments

During this thesis I have learnt a lot about cyber security and have verified my interest. I would just like to thank Prof Ryan Ko for providing me and the rest of his students with the opportunity, as well as the work that has been put into developing the cyber security field at UQ. Furthermore, I would like to thank Mr Taejun Choi and Mr Joshua Scarsbrook for guiding me through the entire process. I know sometimes there was some confusion and difficulties with regards to the University shutting down and the virus, but I think we managed in the end.

I would lastly like to acknowledge Hetong. Hetong was another student completing his thesis along side me on a similar topic. Hetong and I had several discussions after our meetings to which helped realign our direction moving forward with each of our projects. This unfortunately stopped once the restrictions were put in place but was helpful at the begging of the project.

Abstract

As technology and the world develops, devices and hardware are ever increasingly being connected. Cyber criminals are constantly adapting to this, using advanced unfamiliar attacks with malicious intent. The use of an automated defence system would assist in countering these attacks. This thesis acts as a collation of prior art developed for the Cyber Grand Challenge that has otherwise been deprecated some years ago. More specifically, a porting process is presented to assist further development and use of an automated patching tool called Patcherex. This objective is taken further by demonstrating Patcherex functioning within a UQ environment. By automatically inserting a code snippet into compiled executable file, the programs behaviour is slightly changed, forcing the program to print "Hi!" to the terminal display. Presenting this provides a proof of concept for the possibilities of a future UQ specific automated defence system. Where instead of the program printing "Hi!" the program is patching a detected vulnerability. Finally, the actual success of porting Patcherex is determined by running the tool on multiple binaries. Overall resulting in an adequate conceptualisation of the future use of such a system within a UQ environment.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is estimated that “hackers attack every 39 seconds, on average 2,244 times a day” [1] and ”the average time to identify a breach in 2019 was 206 days. [2]. It is becoming increasingly clear that humans alone can no longer defend against such a fast moving and rapidly developing cyber war zone. Large organisations such as UQ are at the forefront of risk with extensive amounts of user data, resources and sensitive information. With a global average data breach cost of $3.92 million dollar (USD) as of 2019 [2], UQ should continue developing new combatant solutions in an attempt to avoid such a costly scenario.

DARPA highlighted this issue on the world stage by hosting the first machine vs machine cyber hacking tournament. The Cyber Grand Challenge (CGC) held in 2016, began with over 100 teams housing some of the top security researchers and hackers in the world [3]. The requirement for both offensive and defensive tools outlined the discrepancies between attacking and defending in human speed vs machine speed. Notwithstanding, it also revealed the considerable amount of work that needs to be done before being able to confidently rely on machines.

1.0.1 Project Outline

The project will review Patcherex, one of the tools developed during the CGC by Mechanical Phish [4]. Since the CGC used a custom operating system and binary format, it was required to port Patcherex to a standard Ubuntu 16.04 Linux ma-

chine hosted by UQ. This porting process is recorded within the report to assist in futher development. The overall result of the project is demonstrated by modifying a compiled binary through the agency of Patcherex, automatically inserting a code snippet into the executable. Furthermore, the performance of Patcherex operating within UQ is analysed by running the tool against multiple binaries with known vulnerabilities.

Overall, providing a proof of concept for future developers to progress. In which, further development of the automation tool will allow UQ to run a self defence system within machine-scale time, ensuring security when cyber-attacks increase in complexity.

Chapter 2

Existing Protection Systems

Before detailing Patcherex as an existing tool, it is important to first review the current defence solutions used by cyber secturity experts. Although endpoints are commonly thought of as a: phone, tv, smart fridge or server, it is actually more realistic to categorize them as a series of programs, scripts and applications running in parallel. Each piece of software has numerous software vulnerabilities as a result of bad programming practices, software limitations or bugs. Thus, as the number of connected devices increases in homes throughout the world the more susceptible devices become to cyber-attacks. This is increasingly relevant as data grows in size and confidentiality.

Of course the protection of sensitive data has always been a prevalent issue, and several counter measure techniques have been developed overtime to increase security of systems and connections. One such technique is the continuous release of patches which fix previously broken/insecure software. Patches are not unique to cyber security and are simply a means of improving software over time.

2.1 Identifying vulnerabilities

In order to patch potentially insecure code, the vulnerabilities must first be identified. Security experts use a variety of existing tools implementing both static and dynamic analysis. These tools typically target a specific operating system, program-

ming language or code type (source code, exe, machine code, etc). In simplifying the broadness of scope, vulnerability detection is more efficient and successful. This is because vulnerabilities vastly differ when considering the aforementioned specifications. Furthermore, each tool focuses on vulnerabilities susceptible to particular attacks.

2.1.1 Identification

Patch identification tools vary in performance results, but strive to achieve a balance between efficiency and accuracy. The results of identification are often ranked due to their false positive or false negative percentiles (generally desired to have low percentiles [5]). To achieve this, there are two main forms of analysis, each with a set of advantages and disadvantages.

2.1.2 Static analysis

Static analysis tools are performed in a non-runtime environment. The analysis typically consists of spanning a given segment of source code or binary objects [6], and matching the observed code snippets with a fixed set of patterns, rules, or behaviours [7]. After probing the source code, static analysis typically provides a break down of flagged results, requiring human observation to make changes [8]. This process can consume time and resources, providing one benefit to creating an automated process.

Although static analysis achieves extensive code coverage and efficiency, its inability to accurately evaluate run time behaviours results in high false positive rates [9, 10]. This often results in the culmination of static and dynamic analysis to enhance performance. [9, 11].

2.1.3 Dynamic analysis

Dynamic analysis adopts the opposite approach by operating during program execution. This allows for analysis of software in which there is no access to the source code [12, 13]. The analysis functions similarly to static analysis, matching run time

behaviours to a set of rules. However, unlike static analysis, program semantics are outlined during program execution (potentially in a virtual machine) and encapsulate the main advantage of dynamic analysis [14].

2.1.4 Improving identification

Before discussing methods to improve analysis performance, a summary of sections 2.1.2 & 2.1.3 is as below.

Table 2.1: Comparison table of Static and Dynamic Analysis

	Static Analysis	**Dynamic Analysis**
Path execution	*Yes (less effective)*	*Yes*
Costly	*No*	*Yes*
Online	*No*	*Yes*
Requires source code	Yes	No
Fast (turn around speed)	Yes	No (comparitively)
Entire code coverage	Yes	No

As seen in the table above, static and dynamic analysis have a relatively antagonist relationship in the sense that dynamic analysis performs in scenarios static analysis fails (and vice versa). Developers and security specialists soon realized the culmination of both analysis techniques increases performance [9, 11]. K. Roundy and B. Miller also implemented this to achieve pre-execution analysis [15]. Allowing them to reduce flagged program vulnerability locations by a hundred-fold (relative to other methods of identifying unpacked code) [15].

Another method of improving identification results is the inclusion of machine learning into the process [8, 16]. Researchers from the Universidade de Lisboa, implemented data mining to minimise the number of false positives detected during static analysis performed on PHP applications [16].The approach explores the use of several different classifiers, evaluating the flagged vulnerabilities identified during static

analysis. Results of the research were successful, achieving overall improvement when compared to other forms of analysis in this space [16]. Similarly, Vuldeepecker is a tool developed with a Bidirectional Long Short-Term Memory neural network to improve vulnerability detection [8]. The article concludes that VulDeePecker substantially out performs both pattern and code-similarity based detection systems [8]. Although the application of machine learning algorithms improves identification, it was not included in this project proposal due to the training time constraints and its sensitivity to the amount of data required. However, future implementation into UQ's self defence system should be considered and explored further.

Yue Chen, introduces a new approach in the form of Clearview, a tool to automatically patch security vulnerabilities in deployed software [5]. The machine learning approaches mentioned above, use data to train a model capable of improving the identification. This differs from Clearview which observes the application's behavior during normal executions to infer a model that characterises those normal executions [5]. Thus, if something is classified as abnormal based on the trained model, it will be flagged by Clearview. This procedure mitigates the requirement of large data sets but still incurs lengthy training times.

2.2 Patching Vulnerabilities

Generally, issues within the software are reported through users naturally finding bugs whilst interacting with the service. Security bugs however require independent testing, as users are not typically attacking the system.

Once discovered, developers will work towards applying a patch manually by releasing an updated version of software. This consists of replacing the identified code with 'fixed code'. Depending on the severity of the breach, some systems have been taken offline whilst fixing. This process is potentially damaging and often costly. An automatic patching system would be capable of constantly monitoring the system [17]. Ideally, identifying vulnerabilities before any malicious user .

Each vulnerability has several known techniques capable of being a solution. One difficulty faced is choosing which solution. Security vulnerabilities have many variables and determining the most effective and efficient solution is crucial [18]. This becomes even more challenging in an automated system as using boolean logic to list all variables approaches impossible as the complexity of the problem grows. Many current researchers propose the solution to this issue is the additional implementation of machine learning [8, 16].

2.2.1 Automatic Patching

If outcomes are ignored, administering the patch is relatively elementary process (simple replacement of code). Normally, this replacement of code would be done via a human modifying the source code of the executable, recompiling, testing, and then releasing the newer version. In an automated system this entire process faces several complications.

When conceptualising ways in which to modify an executable file, from a high level there are two methodologies. Being;

- Insertion of assembly into the binary
- Decompiling, adding code and recompiling

Direct Insertion

If the automatic system directly inserted a code snippet naively, the entire state of the program (including memory) would change [19]. This would most likely cause the program to crash. To navigate around this issue, instead of a direct replacement of assembly code, a technique was developed to reroute program execution away from the vulnerability [20] (see Fig. 2.1).

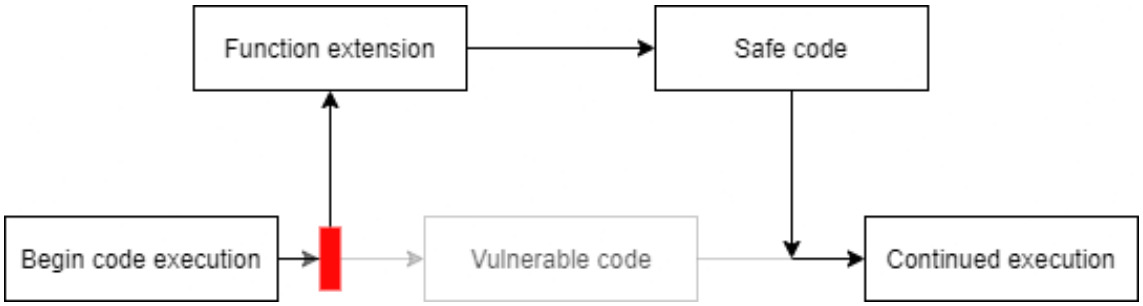

Figure 2.1: Programmatic reroute

This reroute acts as an extension of the original path, permitting execution of new code. For example in terms of buffer overflow errors this reroute could add boundary checks, replace the API or extend the buffer size [11]. By navigating around the vulnerability, the program continues normal execution without removal of any original code.

Recompiling

Initially, this methodology appears simpler than directly inserting into the executable. However, automatic decompilation has many difficulties in itself [21, 22]. These issues may then propagate throughout the code when recompiling. Overall, resulting in a different set of complications. Furthermore, by requiring decompilation and recompilation, this forces the program to be offline. In this thesis, it is desirable to build a system capable of self defence in a run time environment. Thus, tools which use recompilation methodologies were favoured less over tools which directly insert into the executable during run time.

Another complication worth mentioning which applies to both methodologies is the propagation of patches to code clones (frequent code reuse). This issue is particularly relevant in large code bases, and was solved via the use of static and dynamic analysis [9]. For the initial scope of this thesis, code clone verification will be ignored as the demonstration will be executed with a small application. But, should be noted for further development of the system.

2.3 Review patch

After the patch has been applied, it is important to verify its result as it is crucial the patch did not effect the programs behaviour in an undesirable way. Currently, patch revision can be performed with human expertise or a series of automated tests. Human reviewed patches can be time consuming especially if required to set up new environments and initialise different scenarios [23]. Automated testing reduces this time spend considerably. Operating at machine speed, computers can explore an extensive number of execution paths [24]. These reevaluation tests can be executed after patch completion, determining if the applied solution successfully secured the code.

Machine learning could be implemented during this step to improve future results [13, 16]. In Clearview [13] the researchers executed this by constantly refining the machine learning model by continuing to observe the patched application. This was considered out of scope for this thesis but further exploration is recommended.

Actual implementation of an automated testing systems requires determining if the software is still exploitable [25]. Manually, a proof of vulnerability is as simple as attempting to reexploit the vulnerability. This process can be automated by executing a list of instructions. DARPA and Mechanical Phish developed a novel way of structuring these proof of vulnerabilities (POVs) into a consistent format [26]. This made testing the binaries with automatically generated POV's much easier (see Section 2.6).

It is proposed that the tool developed for UQ implements automated testing, keeping the entire system automatic. However, an appropriate manual revision process should be considered to verify the correctness of each patch. This procedure should mirror that of a mass production line, verifying correctness every x number of patches. Furthermore, it would be beneficial for UQ to model the aforementioned POV's used in the CGC. Allowing the system to scale well by keeping a universal testing POV between automatic exploit generation and automatic patching.

Improving patch results

Similarly to identification, machine learning algorithms can be used to enhance patch selection. In turn increasing the effectiveness of applied patches [13]. This allows the patching tool to select which patch to apply depending on the classification of the identified vulnerability. Hence, would only be possible if a combination of patching tools or patches were used. Although this is the desired end goal of this thesis (future developers adding tools to framework) this feature was not implemented in this thesis.

2.4 Cyber Grand Challenge

The Cyber Grand Challenge (CGC) [3] held on August 4, 2016 challenged this idea by hosting the first machine vs machine hacking tournament. The challenge was designed to encourage the creation of automatic defense systems, that were capable of automatic detection, patching and exploitation. By operating during run time on a machine scale, the tools developed and theorised during the CGC challenge today's black-hat dominated cyber warfare space.

2.4.1 DECREE

DARPA Experimental Cyber Research Evaluation Environment (DECREE) was the operating system created during the challenge [27, 26]. DECREE was designed to remove the complexities of a modern day operating system and to include a number of features making it an ideal platform for experimenting with security vulnerabilities. The most important of these features being simplicity and high determinism

Familiar operating systems such as Linux, Mac and Windows have hundreds of native system calls, most completely unique and incompatible with each other. DECREE was designed to have just seven (see Appendix A). This limits the complexity when attempting to automatically detect and identify vulnerabilities, allowing competitors to focus on program analysis techniques [26]. Due to this simplicity, DECREE also requires its own executable format. These executables only have a

single entry point method, reducing the barrier to entry to begin the development of automation tools [26].

Deterministic systems follow a strict set of rules such that given a specific state or input, the same output will be produced, regardless of how many times the program is run [28]. Although perfect state replay is near impossible, DECREE's predictability allows programmers using the DECREE system to structure their tools with some degree of confidence that the operating system will function as expected. This predictability property has been enforced all throughout the DECREE platform stack, as well as the kernel's core functionality [27].

Unfortunately, the same features which make DECREE such a great operating system to experiment with, also isolate the tools from familiar operating systems. Meaning, DECREE's custom built operating system and binary format, differs so much from modern day tools that a lot of the code and protocols developed for the Cyber Grand Challenge are incompatible with existing software and analysis techniques. As well as this, although DECREE was open sourced, the project has been discontinued and community support is limited, making it difficult to adopt and port the software.

2.5 Patcherex

Patcherex was the automatic patch insertion tool developed by Shellphish during CGC [26]. Like DECREE, the project was open sourced and released to GitHub after the challenge [4]. The tool was built on top of Angr, a Python framework used to analyse binaries [29]. Patcherex provides several methods of insertion into binary files, as well as accompanied vulnerability detection techniques. Structured around 3 main concepts, the core functionality of Patcherex stems from it's;

- Patches
- Techniques
- Backends

This thesis deconstructs the existing tools Patcherex offers, in an attempt to gain a detailed understanding of how UQ could further develop and integrate Patcherex within UQ's systems. A detailed explanation of each of these components is introduced below.

2.5.1 Patches

Patcherex patches define how the assembly should be inserted. Patcherex provides several different patches where a patch is defined as a single modification to a binary [4]. A full list of patches along with brief descriptions is as follows.

Table 2.2: List of Patcherex patch classes

Patch Name	Description
CodePatch	Base class for all code patches
InsertCodePatch	Specifies code to be inserted before an instruction at a specific address
AddEntryPointPatch	Adds code instructions before the original entry point of the binary
AddCodePatch	Utility function which adds string literal code other patches can use
AddRWData	Utility function which adds read-write data other patches can use
AddRWDataPatch	Inserts read-write data before an instruction at a specific address
AddRWInitDataPatch	Adds read-write data to the initiation of the binary
AddLabelPatch	Adds a label at a specific address
AddRODataPatch	Adds read-only data at a specific address
RawFilePatch	Uses a another patch to insert read-write data at a specific file address
RawMemPatch	Uses a another patch to insert read-write data at a specific memory address
SegmentHeaderPatch	Uses the **AddSegmentHeaderPatch** to insert code instructions into the segment headers
RemoveInstructionPatch	Removes an instruction of size x at a specific address
PointerArrayPatch	A patch used to compress pointer arrays

Some of the patches are used as tools for other patches to reference [4]. This provides a modular framework to build complex patching methods by combining several patches.

2.5.2 Techniques

A Patcherex technique refers to a methodology applied to a binary file which will analyse and detail specific vulnerabilities. These techniques are interchangeable, ideally running all techniques against any given binary file, which will return a list of patches for each, and give the best chance to defend against all possible exploits.

Adversarial attacks attempt to reprogram certain parts of an executable, such that the program will perform a task chosen by the attacker [30]. Patcherex introduces an adversarial technique, directing any suspicious traffic to a function pointer which runs an infinite loop of memory allocation, eventually crashing the program and continuing normal execution.

Backdoor attacks refer to unauthorised users attempting to gain authorised access to a machine. Backdoors are often deliberately placed into a system, as a means of gaining full control in case of an emergency. Patcherex deals with backdoor attacks by applying a patch which exposes a 'fake' backdoor that shadows the real backdoor. This acted as a honey pot, attempting to trick potential attackers into thinking they had successfully found the backdoor [31].

Binary Optimisation is a principle of computer science optimisation commonly reffered to as constant propagation optimisation. The idea is to remove calculated constants after program execution, optimising performance by reducing the number of calculations made. Although this is not really a patch, Patcherex provides this technique as it reduces the amount of information/clues an attacker might gain during reconnaissance. As well as, free up computing power that can be used for other processes.

Simple Pointer Encryption decreases readability of the program by looping through the executable and encrypting any pointers that are found. Depending on the complexity of the encryption, this may also decrease performance of the program [4]. Patcherex approaches this technique by first clearly labelling a set of

definitions. These definitons are the instructions which complete the following tasks;

- Pointer source - loads a pointer into memory or immediate
- Pointer consumer - referencing a pointer
- Pointer sink - writing a pointer to memory or another register
- Pointer transformer - incremenents/decrements a pointer and its output register

It is guaranteed that all pointers comping out of the source are encrypted [4], decrypted before reaching the pointer consumer and rencrypted before reaching the pointer sink. Patcherex implemented this as the simplest solution, however it is sub optimal in several cases such as instances of multiple pointer consumers/sources, as well as when the pointer sink is empty.

Bitflip technique was developed by Patcherex to confuse their competitors. By flipping the bits of each binary, other teams automatic exploit engines may fail if they are not generalised enough. Developed specifically for the challenge binaries, Patcherex admitted this technique did not function as initially intended [4, 26]. Furthermore, it contains a known bug that causes the patched program to crash.

CPU ID inserts a piece of code into the executable which attempts to identify each cpu, blacklisting any 'unknown' cpu operations. This patch technique is a preemptive defence technique as oppose to a reactive patch.

Fidget [32] is another tool built on top of Angr [29], allowing easy integration with Patcherex. Fidget aims to analyse a program, report its functions in a structured layout and modify stack frame allocation in an attempt to mitigate any attack which makes general assumptions about the stack layout. The Fidget patches claim to "protect against any exploit leveraging a stack buffer overflow or format string vulnerability" [32]. It functions by first reading the stack and classifying memory allocations into variables. These variables are then converted into satisfiability modulo

theory (SMT) [33] constraints and parsed into z3 [34, 35] (SMT solver), resulting in a different ordering of the original layout without modifying functionality. In lamens terms, reordering the programs memory allocation without modifying the programs original behaviour.

Shadowstack is a mechanism used in computer science to protect and verify procedure return addresses stored on the stack [36]. By shadowing the program call stack, an overseeing function can detect attacks by comparing the original stack and the shadow stack. Any difference in the stacks will trigger an alert or the next phase in patching the program. Monitoring a shadow stack is a common technique relating to maintaining control-flow integrity. To reduce complexity and increase efficiency, Patcherex ensured that the shadow stack only stores the original return addresses as opposed to all variables, arguments, etc [4].

Indirect Control Flow Integrity (CFI) protects programs during run time by redirecting the flow of execution [37]. There are many common CFI techniques such as code-pointer separation (CPS) [38] and shadow stacks [36]. When viewing the github repository, the technique is not completely finished and is littered with TODO's.

Simple Control Flow Integrity is another implementation of CFI protection. This version was developed as a reduced version of Patcherex's unfinished Indirect CFI. This technique essentially aims to fix the same issue as Indirect CFI, however, it is less scalable and is not as modular [4].

Malloc Extension Patcher is a useful technique which targets common attacks such as buffer overflow exploits. Patcherex analyses the executable and locates any malloc or realloc function calls. It then applies a templated malloc extension patch to each of the identified locations. This patch consists of padding one or both sides of the originally allocated memory [4].

No Flag Printf was developed specifically for the CGC and is not vary useful for external use. However, it's aim is to detect an attack on a known flag page location in memory and prints a message back to the attacker.

Non-Executable Stack (NxStack) protection refers to setting the NX bit, blocking an attacker from executing a functions whom's return address value points back to the stack [39]. This technique will ignore any functions which are syscalls or sim procedures, and thus, should be used in conjunction with other techniques.

Packer patch is a generic technique which compresses the executable. This technique should be applied to any binary without concern. The idea of this patch is to reduce the potential exploits of the binary.

QEMU is an open sourced machine emulator which allows virtualisation of machines [40, 41]. Patcherex used a custom QEMU during the CGC, designed to emulate a DECREE system to run isolated tests on specific binaries. **QEMU Detection** was developed as another identification technique which establishes whether functions/operations are 'safe' by acknowledging if they are being called from within the same QEMU.

Random Syscall Loop patch was designed to deter/confuse potential attackers. Patcherex aimed to confuse the opposition teams automatic exploit engines by inserting random system calls. This technique could be applied to binaries outside of the CGC but the effectiveness would have to be explored.

Stack Return Encryption iterates through the stack and determines a list of unsafe functions. In this case, Patcherex determines a function as unsafe if it fails the following checks;

- The number of function calls is greater than a threshold (adjustable but default is 5)
- If the function is a syscall, it is not a "recieve" syscall (see Appendix A)

- The function is called by printf or malloc

Once a function is listed as unsafe, Patcherex encrypts the return address on the stack in an attempt to obscure the programs behaviour.

Transmit Protection is designed to safeguard any outgoing transmissions by placing it inside a custom wrapper. This wrapper was designed by Patcherex, allowing them to identify safe transitions.

Uninitialised Patcher was designed to assist other patching techniques. The uninitialised patcher maps all block addresses to each written or read register. Once this map is constructed other patches can quickly reference it if need be. This saves computational power by avoiding multiple patching techniques creating individual maps.

2.5.3 Backends

Once a technique has analysed an executable and reported a list of patches, Patcherex inserts these patches by using one of two backend components. These backends allow insertion of assembly language string literals into given locations, and are summarised below.

- Detour Backend inserts a given patch into the original code by using a jump out and jump back technique. This method moves a single function out and extends all other functions through this function.
- Reassembler Backend first decompiles the original executable, inserts the the patch and then reassembles the binary.

Although each of the backend components have specific use cases, to limit the scope of this thesis, only one was chosen. To help decide which backend to focus on, a comparisons list was constructed and is shown below.

Table 2.3: Comparison table of Detour and Reassembler Backend Components

	Detour Backend	**Reassembler Backend**
Requires compiler	*No*	*Yes, furthermore, the Reassembler Backend was originally designed for DECREE binaries which require a custom compiler. This means that the Reassembler Backend will only function on DECREE binaries used with Compilerex, Patcherex's custom compiler*
Optimised	*Generates bigger and slower binaries, and in some cases cannot insert some patches*	*Results in cleaner more efficient binaries*
Requires decompilation	*No*	*Yes, Compilerex also has a built in custom decompiler, however, an external tool would need to be used to function on native x86 binaries [21]*
Reliable	*Patcherex disclosed that using the Detour Backend results in less broken binaries when compared to the Reassembler Backend*	*No (comparatively)*

From the table above, the Detour Backend was chosen to be the focal point for this thesis. This is largely to due to the complexity of compilers in terms of understanding compilation and decompilation. Focusing on the Reassembler Backend would require reconfiguration of the tool to function with common compilers such

as gcc, which have a lot of existing security features [42, 21]. This would increase the difficulty of the task at hand considerably. Furthermore, since the goals outlined in this thesis was to provide a proof of concept, it was deemed that the reliability of the tool was paramount. However, it should be noted that the Detour Backend still has several flaws and requires further development.

2.6 Proof of Vulnerabilities

A proof of vulnerability (POV) is used to demonstrate an exploit is possible against a given target. Similar to a proof of exploit or proof of concept [25], POV's assist in disproving a false positive, or affirm that the vulnerability exists. POV's generally require full simulation of the actual attack as to confirm the exploit. This process can be automated by scripting a set of instructions against a particular target.

During the CGC a POV was a specific binary program which aimed to do two things;

- Cause a segmentation fault
- Read flag data from a specific memory address

If running the POV with an opponents binary caused either of the above, the exploit was considered successful [26]. POVs provided a simple and uniform way of verifying vulnerabilities and testing. It is proposed that UQ's automated defence imitates a similar protocol. This would require the setting of what is considered flag data. For example, before code compilation, session keys, passwords or any other sensitive information would be defined as 'flag data' and stored at a specific memory address. The POV will then attempt to access it by exploiting a vulnerability.

2.7 CB Multios

A substantial set of challenge binaries (CBs) was custom made during the CGC, with each binary relating to one or more vulnerabilities [43]. This set contained binaries with a broad variety of exploitable flaws. Like all other software developed

for the DARPA challenge, these binaries did not function on operating systems outside of DECREE. Fortunately, an open source project modified the CBs, making them compatible for use within a Linux environment and existing analysis tools [43]. These executables were used to demonstrate Patcherex as well as test Patcherex's overall performance. This repository also ports the extensive tools associated with the CBs such as performance monitoring tools and functionality tests. These tools were not used for this thesis but may be useful in further development.

Chapter 3

Theory

3.1 Porting Software

Porting refers to the methodology used to adapt software to function within an environment differing to the one it was originally written in or designed to operate in. This term is also used synonymously when referring to required hardware upgrades when moving to a new environment. Porting, is a very common process largely driven by user demand on multiple operating systems. It is required as each development environment has a fairly unique set of memory management rules, system calls etc, which effect the behaviour and functionality of all components within it [44].

3.2 Ghidra

Ghidra is a software reverse engineering (SRE) framework original developed by the National Security Agency (NSA) of the United States [45]. The project was open sourced in March 2019, making the tool accessible to all.

The software provides and extensive suite of SRE tools, but is mainly used in this project to identify memory locations of specific function calls. Ghidra also provides a usable API offering potential to further integrate Ghidra into UQ's automated defence system.

3.3 Memory Registers

Within a computer there are a number of memory addresses and registers which are directly accessible by the processor. The exact number of addresses is determined by the size of RAM and CPU (32-bit or 64-bit). Some memory addresses are reserved and accessible with defined keyword labels.

Important registers

Later in this report, several assembly language code snippets will be introduced. These snippets reference some of these memory addresses. The list below briefly describes each memory address, providing context and general use. Each of the below registers is 32-bits [46]. A full visualisation of how the registers are divided can be found in Appendix. B.

Table 3.1: Description of Used Registers

Register Name	**Volatility**	**Description**
eax	*Non-volatile*	*Known as the primary accumalator. Generally used as a utility register for input and output. For example, in arithmetic operations it might store one of the expressions operands.*
ebx	*Non-volatile*	*Known as the base register. Often used when index addressing.*
ecx	*Volatile*	*Known as the count regiser. Used to store the count of iterations when looping operations.*
edx	*Volatile*	*Known as the data regtser. Similar to eax, this register is generally used to store input and output data. Sometimes this register is required if the value cannot be completely stored in eax (data over 32 bits).*
esi	*Non-volatile*	*Known as the source index register. This register can be used as a general purpose index register however it does have unique operations when combined with special opcodes. These opcodes efficiently move data from the esi (source register) to edi (destination register) described below.*
edi	*Non-volatile*	*Known as the desination index register. Partners the esi register. Similarly to the esi register, this register can be used as a general purpose index registers.*

Where in the above table volatility refers to how stable the register is in terms of keeping its value. For example, the esi register generally stores data throughout a function as it its value will not change unless forcefully overwritten.

3.4 Opcodes

Opcodes refer to a collection of low-level machine readable operations. These codes are the level above binary instructions [47]. Some opcodes will be referenced later in the report as they are the instructions that will be inserted into an executable to perform specific tasks. The following opcode descriptions provide context to subsequent references.

Table 3.2: Description of Used Opcodes

Opcode	**Description**
PUSHA	*Pushes all general purpose registers to the stack. Where the general purpose registers include those mentioned in 3.1 [48].*
POPA	*Adversely to pusha, popa pops values from the stack into the general purpose registers.*
RET	*Returns to the address located at the top of the stack. Generally, this address is placed on the stack by the CALL opcode*
CALL	*Branches to a target operand after pushing procedure linking information to the stack. The target operand can be a general purpose register, immediate value or memory location.*
INT	*Used to generate a software interrupt. Parsed an interrupt number as a hexadecimal value.*
MOV	*Moves one operand to another. Takes two operands as arguments, destination and source respectfully. Moves the source operand to the specified destination. Both operands can either be a general purpose register, immediate value or memory location.*

Chapter 4

Environment Setup

4.1 Initialising UQ zone

The first step of demonstrating use of an automated patching tool, was to instantiate an appropriate environment. This environment had to be suitable for existing systems, as well as consideration to future systems. A UQ zone was instantiated with 64-bit Ubuntu 16.04.02. Ubuntu is a widely accepted and used Linux based operating system and has been continued for many years. The requirement for this tool to be demonstrated on Linux was trivial as it powers 96.5% of the top one million domains [49]. Ubuntu 16.04.02 was not the newest version at the time of creation, however, it is cross compatible with all newer versions. Furthermore, more consideration was put into the environments memory management, rather than the OS version. A 64-bit system was chosen over older 32-bit systems, as 32-bit systems will become obsolete in future years to come. The exact specifications of the instantiated zone are can be found in Appendix C, and an exact guide can be found in UQ's zone management documentation [50].

4.2 Package Collection

Once an appropriate zone was created, the process of porting Patcherex from a DECREE environment began. The project did not get very far before being confronted with a large list of required packages. The majority of essential libraries are depre-

cated and discontinued, thus, making it a slow and tedious process of scraping the web to find them. Fortunately, the packages were eventually found and the correct environment can be created with the following steps. First, ensure the following apt packages are installed.

```
apt-get update && sudo apt-get install nasm clang git python3-pip
```

A Python virtual environment must also be created and correctly activated before attempting to install any packages. Some libraries can simply be installed with pip, however, the following packages in table 4.1 must be installed manually by cloning the repository and running

```
pip install -e . && python setup.py install
```

within each cloned directory. This adds the package to your virtual environment. In this process order must be maintained as there are some dependencies between the libraries.

Table 4.1: Repositories for Deprecated Libraries

Library Name	**Command**
Povism	*git clone https://github.com/mechaphish/povsim.git*
Fidget	*git clone https://github.com/angr/fidget.git*
Compilerex	*git clone https://github.com/mechaphish/compilerex.git*

Once the above packages are installed, Patcherex should have some immediate functionality, this can be demonstrated by following the steps provided in the github README [4].

Chapter 5

Patcherex

Once the environment and all required packages were setup, the next stage of the project was to demonstrate use and functionality within the UQ system. This chapter details the porting of Patcherex as well as demonstrations and examples of uses.

5.1 Porting Patcherex

Since Patcherex was initially designed for a specific custom binary format and to operate in a DECREE (see Section 2.4.1) environment, porting was required for functionality within Linux. The overall process required a slight modification to the build system and the removal of DECREE specific functions and system calls.

5.1.1 Removing unnecessary function calls

The first step was to identify any unnecessary function calls as there is no point in porting software which will not be used in the future. Furthermore, this reduced debugging complexity as removing functions limited the number of variables which could cause the program to crash. To begin with Patcherex's Detour Backend was targeted as it was considered the most critical component to be ported.

During the cyber grand challenge each of the challenge binaries came with an associated pdf which detailed a description of the the functionality of the program. Thus,

when running either of the backends, Patcherex attempts to find a PDF within the executable and remove it. This was not necessary for the initial demonstration so the code that performed this was removed. This removal was noted with comments and a link to a previous version was left in case this functionality was ever needed again. No other functions were completely removed.

5.1.2 Debugging process

The next stage of porting was a bit more undirected and required use of external tools to assist debugging. For this stage of porting, GNU Project Debugger (Gdb) [51, 52] and Python Project Debugger (pdb) [53] were used. This helped identifying exact point of failures within the program by stepping through the executable.

Pdb and Gdb

Pdb is a built in Python lib module, providing an interactive Python source code debugger. The module provides many useful tools such as but not limited to single line stepping, conditional breakpoints and the inspection of stack frames [53]. At some points during this progress the program failed due to hung processes. Pdb can not support debugging these errors as it does not provide a Python traceback. In this case gdb was used instead. On top of this, gdb proved additionally useful as it caters for segmentation faults which are uncaught in Python's exceptions [51].

Common issues

Since the DECREE operating system was a simplified version of a Linux 32-bit system, a lot of the issues stemmed from overwriting memory or attempting to access restricted memory. In practice this essentially meant modifying any defined memory size/offsets from 32 to 64. This was done very carefully as to not extract any desired functionality.

A similar issue extended into the patch detection techniques. In this case some of the defined common patches applied by the techniques were incompatible with

systems other than DECREE due to the way Patcherex searched and defined free and available registers.

5.2 Using Patcherex

There are two ways that Patcherex can be used in practice. Each procedure can be more/less benefical depending on the use case. The first approach is to use a defined Python class called Patch Master [4]. This method is a quick and simple way to gain access to the entire Patcherex library but should not be used arbitrarily without understanding each component of Patcherex with some degree of detail. Using this approach as the underlying framework to a system may cause future issues if the lower level components are not completely understood.

The second method is simply combining each of the Patcherex components to fit a more specific use case. By treating each Patcherex component as an interchangeable building block of the system, new or modified components can be integrated into the underlying architecture.

5.2.1 Using Patch Master

As reffered to in Fig. 5.1, Patcherex designed an easily usable class called patch master. By combining all components of Patcherex, the module can conveniently

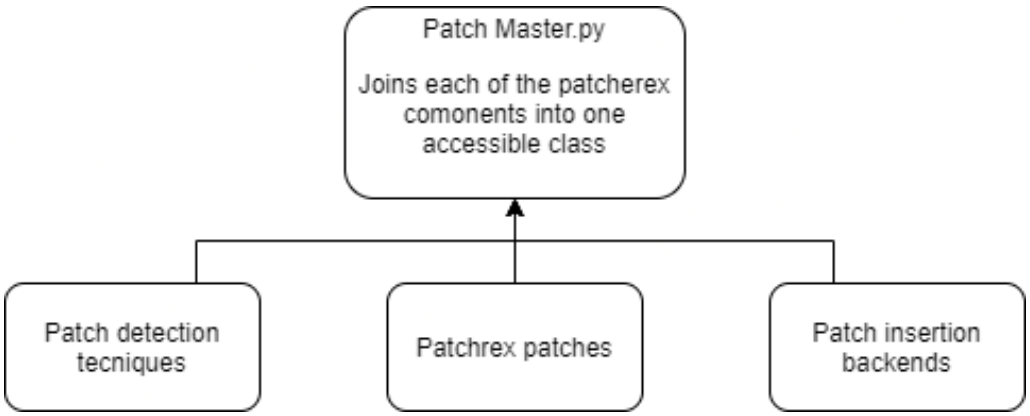

Figure 5.1: Patch master class structure

be used by calling

```
./patch_master.py <run_type> <input_file> <method> <output_file>
```

Where in the above usage the option "run_type" can be instantiated to; run, single or multi.

Run type

Run being the simplest options, executing the script with no subsequent tests, single running the script on an input file, using a single specified method (see Section 5.2.1) on an input file and then testing the result, and multi which runs the scripts using multiple methods and tests the results.

Input and output file

Trivially the input_file option refers to the string literal path leading to an executable binary, and the output_file is the path to which Patcherex will write the resulted 'patched' binary.

Methods

Methods determine which technique (as listed in Section 2.5.2) will be used to identify potential vulnerabilities. Patch master does not support all techniques defined in section 2.5.2. Initially only providing two methods, which support the stackretencryption and bitflip techniques.

Fortunately, writing new methods to accommodate other techniques can be done by adjusting the following defined template. Where "technique reference name" can be defined as any string, this string will be used as the "method" parameter when calling patch_master, and "technique name" must be an imported Patcherex technique class (see Section 2.5.2).

```
def generate_<TECHNIQUE REFERENCE NAME HERE>_binary(self, test_bin=None):
    backend = DetourBackend(self.infile)
    patches = []
    patches.extend(<TECHNIQUE_NAME>(self.infile, backend).get_patches())
```

```
    backend.apply_patches(patches)
    final_content = backend.get_final_content()
    return final_content
```

Since all techniques require a get_patches method, this template is mostly universal, with the exception of some techniques requiring additional function calls.

5.2.2 Integrating patcherex from the ground up

Using individual Patcherex components to build a custom framework forces the developer to have an underlying understanding of each component. It should be noted that taking this approach does not mean patch_master must be completed disregarded, it simply encourages better programming. Simply using patch_master without insight into what the program is actually doing will only cause further problems in the future. As seen in Fig. 5.2 using Patcherex's existing tools is not excluded, rather integrated alongside additional tools. It is proposed that this approach is a better way to develop UQ's automated patching tool, as opposed to solely using patch_master.

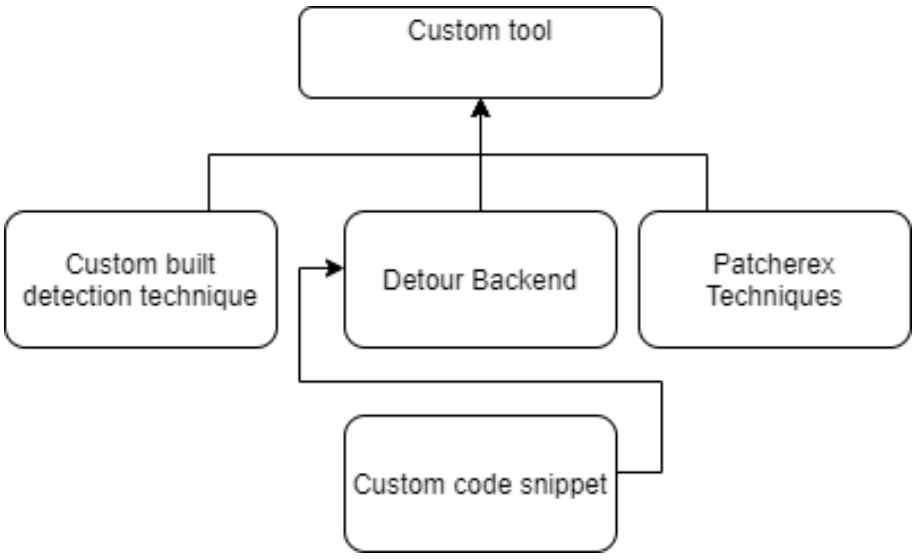

Figure 5.2: Custom tool integrating Patcherex

Furthermore, Fig. 5.2 demonstrates the use of custom code snippets being inserted via the Detour Backend for specific use cases. This will become increasingly relevant as technology advances so quickly, the requirement to quickly modify patches is paramount.

5.3 Testing Patcherex

When beginning or integrating any software project, it is important to implement a structured testing framework to verify functionality. Patcherex defines several testing functions that validate the patch techniques by comparing the outputted binary with a POV (Section 2.6). Unfortunately, all existing tests provided in the Patcherex library were designed for DECREE binaries, and fail if attempted to run against Linux binaries. This means that a new testing framework needed to be established. This new proposed testing framework models the existing framework, differing only in POV's and and POV simulations.

5.3.1 Existing Patcherex tests

Patcherex provides tests for all built components (see Section 2.5). As mentioned previously these tests run using binaries and tools specifically designed for DECREE and the CGC. Executing these existing tests is as simple as running a Python script. Furthermore each of the tests default to run all tests, but, provide the option to provide a list of arguments corresponding a specific set of tests to run.

Backend tests

The backend tests are designed to verify successful insertion of patches into several provided binaries. These tests ignore the correctness of the patches and consider success as no errors. For example, if running the commands

```
var = apply_patches(patches)
var.save(target_filepath)
```

does not return an error and therefore, var is not null, the test determines the backend to be behaving normally. This is not a full test solution as it does not

confirm patch correctness. However, patch correctness is confirmed in subsequent tests and thus, successful insertion is the only thing necessary for these tests.

Technique tests

As stated above, these tests aim to verify the correctness of the patch. As well as this, they allow testing of the technique itself. Confirming successful detection of the target vulnerability, and application of the appropriate patch. This is achieved through the simulation of POV's using Povsim or checking memory register values at different stages of execution. Povsim is a POV simulation library developed by Patcherex for the CGC [54]. Unfortunately, this once again means Povsim is incompatible with external systems outside of DECREE.

This test suite houses a test for each technique defined in Section 2.5.2. The test structure detailed in Fig. 5.3 provides a basic concept for all defined tests.

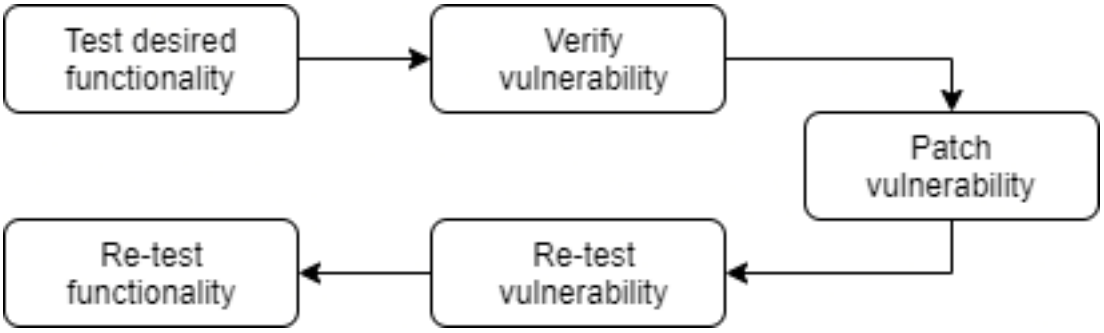

Figure 5.3: Technique testing structure

The first step of the testing process is to test the desired functionality. This phase differs quite significantly between each of the tests as each of the binaries are programmed to complete different tasks. In general, testing functionality can be completed similarly to standard unit tests, by supplying input and asserting the output against an expected output.

Verifying the vulnerability is a bit more complex and requires the use of Povsim as mentioned above. Povsim simulates an exploit against the given binary using a provided POV. Not all tests require the use of Povsim and some vulnerabilities can

be verified with a different methodology. This mainly consists of manually checking memory addresses at specific locations. For example, the testing of transmit protection uses a modified QEMU runner and asserts on specific register values as the program is running [4].

Once original functionality is determined and the vulnerability is verified, the testing binary is analysed, resulting in a list of patches (see Fig. 5.3). These patches are then inserted into the binary, and reflected in the outputed binary. It should be noted that in most technique test cases written by Patcherex the Reassembler Backend is used. Since this thesis focuses on using the Detour Backend, a new wrapper was defined. This wrapper closely resembles the Reassembler Backend wrapper given by Patcherex.

After applying the appropriate patches to the binary, it is important to rerun the original tests. This certifies two constraints which define success;

- The programs functionality should not change
- The vulnerability should no longer exist

If the result of rerunning the tests is as stated above (provided the vulnerability existed originally) then the technique is functioning as desired.

By combining both the technique and backend tests, Patcherex provides a full testing suite for their tools. However, due to the use of external tools such as Povsim which was designed for DECREE systems, a new testing frame work is proposed in Section 5.4 below. This new framework closely resembles that of the existing framework, only remodelling/removing all tools specific to DECREE.

5.4 Future Testing

It is proposed that UQ develop its own POV simulator, designed to fulfill the same purpose as Povsim. This new tool could be developed by adhering to the same

porting process adopted in this thesis and outlined in Section 5.1. This task was out of scope for this thesis but could serve as a future project and is required at some stage if a UQ specific automated patching tool is to be further developed.

5.5 Demonstrating Patcherex

A trivial task was designed to demonstrate Patcherex functioning on a GNU compiled binary [42] in a non DECREE environment. The purpose of this exercise was to exhibit the Detour Backend successfully inserting a piece of assembly language into a compiled binary.

The approach began with choosing a simple Palindrome executable. When running this program the user is continuously prompted to enter a string as standard input. Once a string has been entered, the program processes the string and determines if the input was/was not a palindrome. The simple goal of this exercise was to locate the function call which accepts the user input. At this point, the Detour Backend will insert a code snippet which prints the string "Hi!" to stdout. An illustration of this can be seen below. Fig. 5.4.

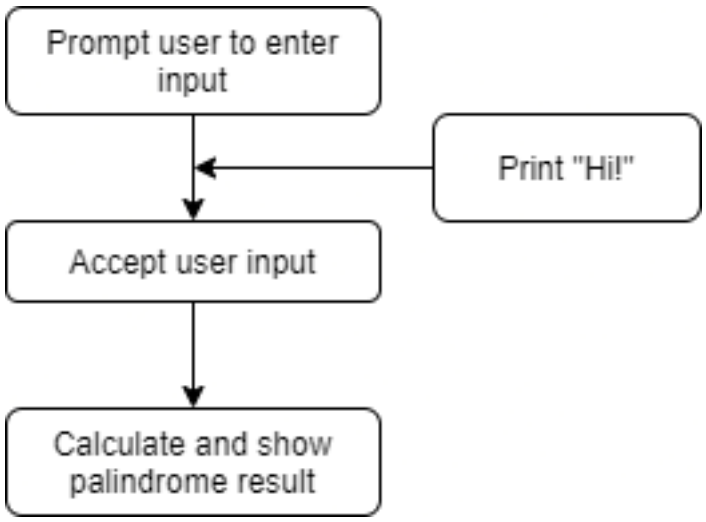

Figure 5.4: Demonstration high level concept

The result, a visual representation of the modified binary such that each time the user enters input, the string "Hi!" is shown in the terminal. Although this is only

demonstrating simple execution of the Patcherex tools, the concept applies to all code snippets/patches. Furthermore, the significance of this example is a working presentation outside of a DECREE environment.

5.5.1 Locating the input address

Ghidra (introduced in Section 3.2) was used to locate the function call responsible for accepting user input. Ghidra provides a modest interface, allowing direct comparison between assembly language and recompiled source code. In this case the original Palindrome source code was also readily available so reengineering through Ghidra was unnecessary. The tool was only used for direct comparison and visualisation of memory allocation. Once the exact address was located, the required code snippet could simply be inserted as the next function call.

5.5.2 Demonstration code

The demonstration code can be found in test.py. For this, we introduce a transmit function as seen below. Refer to Section 3.3 for more information regarding specific memory addresses.

```
transmit_code = '''
      pusha
      mov ecx,eax
      mov edx,ebx
      mov eax,0x2
      mov ebx,0x1
      mov esi,0x0
      int 0x80
      popa
      ret
'''
```

This transmit function is called after loading the required "Hi!" string into the eax buffer. The purpose being to print whatever is in the buffer and return back to the

original program.

This function is added to the list of patches by caling Patcherex's patch constructor.

```
patches.append(AddCodePatch(transmit_code,name="transmit_function"))
patches.append(AddRODataPatch("HI!\x00",name="transmitted_string"))
```

As seen above the raw string data is also added as a patch. This is useful as patches can refer to another patch address by putting its name between curly brackets. Use of this is shown below by creating a new patch and referencing the two patches created above.

```
injected_code = '''
    mov eax, {transmitted_string}
    mov ebx, 4
    call {transmit_function}
'''
```

This code is then directly placed into the binary by calling.

```
insert = InsertCodePatch(location,injected_code,name="injected_code"))
patches.append()
```

Where location referenced above is associated to the identified location accepting user input (see Section 5.5.1).

The result of packaging and running the above code is a replica binary to the original Palindrome executable, only now "Hi!" is printed whenever user input is received. As mentioned previously, this demonstration does not have any usable significance. However, the applications possible using the same process is endless. For example, the "Hi!" string could be replaced with a defined buffer overflow patch.

5.6 Further Exploration

Moving towards the goal of fully autonomous patch insertion, as opposed to demonstrating Patcherex functionality, an exploration into Patcherex's techniques was be-

gun. The goal of this investigation was to further test Patcherex's functionality by executing different techniques on many different binaries not just the Palindrome program. One limitation of this exploration was limited access to POVs. Without POVs for each of the binaries, the actual success of the patch insertion could not be quantified. Thus, success was requantified to - the successful execution of insertion of patches into the binary, with no breaking errors.

For this, a subset of techniques were tested on all working cb-multios binaries (see Section 2.7 [43]. A subset of techniques was chosen due to time constraints and included the following;

- malloc extension patcher
- stack return encryption

The task was completed by creating a simple script which looped through all executables listed by cb-multios and parsing them as arguments into patch_master. This resulted in a total of 484 patch_master calls (242 executables * 2 techniques), with the results being recorded in a log. A full analysis and discussion of the results is shown below in Section 6.1

Chapter 6

Results and Discussion

The results of the thesis is two fold. The first of which is immediate results. The functionality of Patcherex operating within a system other than DECREE, is directly shown through modifying a programs output. Without having to recompile the binary, the Detour Backend was successfully inserted a code snippet which slightly altered the programs behaviour. This provides a clear indication to the potential possibilities of UQ implementing their own automated defence system.

6.1 Exploration Results

The second portion of results, refers to the analytics gathered while testing Patcherex's functionality on several different executables with multiple techniques (see Section 5.6). A summary of results is as seen in Table 6.1.

Table 6.1: Success Rate of Running Patcherex on Multiple Executables

	Successfully Executed	**Failed Execution**
With Found Patches	*2*	*206*
Without Patches	*213*	*63*

Out of the total 484 calls to patch_master, only two successfully inserted a patch into the binary without failing during Execution. This could be due to a number of reasons discussed in Section 6.1.1. Ineffectively, 213 runs were successful without applying patches. Unfortunately, this provides little insight into the success of the

program they were silently successful. Of the failed runs, 206 failed whilst attempting to apply the patches to the backend, and 63 failed during the analysis/detection phase.

6.1.1 Successful executions

Overall, 44% of attempts were successful. Although this seems like a promising result, a large portion of these successes were runs which did not detect any patches. This suggests that the technique failed to find any vulnerabilities within the binary, and there is no means of knowing if these runs would have been successful if patches were found. However, due to the high percentage of failed execution with found patches (≈42%), it can be assumed that of these 213 success' a high percentage would have failed if patches were found. Furthermore, only two runs finished execution with found patches. Both of these runs were executed on the Palindrome binary mentioned in Section 5.5. This suggests the porting process was over fitting and far to specific. And, is highly likely, that these two successful runs targeted the Palindrome program tested on previously (one successful run for each method).

6.1.2 Failed executions

Contradictory to the above results, 55% of patch attempts failed. These failures are categorised into with and without found patches. The 63 failed runs without patches, suggests the program failed whilst using the technique analysis tool. From further investigation a few useful observations were made, these are detailed in Table 6.2 below. Further explanation of these results can be found in Appendix D.

Table 6.2: Failed Patch Attempts

	Stackretencryption	**Malloc Extension Patcher**
Failed during analysis	*38%*	*62%*
Successfully found patches	*64%*	*21%*
Failed due to memory bus error	*100%*	*100%*

The first metric shows out of all failed executions, 62% were caused by the Malloc Extension Patcher, and 32% by the Stackretencyrption technique. This is interesting as both techniques were ported using the same process, making it difficult to draw any concrete conclusions. It is suggested that the Malloc Extension Patcher has several failing paths which are not always executed. Meaning, on occasion, the program runs a successful path depending on the boolean logic specific to each binary.

Secondly, out of all attempted patch runs, the Malloc Extension Patcher only identified potential patch locations 21% of times. This differs from Stackretencryptions 64% and is dependent on two potential factors. The first factor being the classification of how many tested binaries called malloc. The Stackretencryption is a generalised patching technique applied to all functions. Whereas as the Malloc Extension Patcher attempts to find malloc calls. Thus, if the tested binaries do not call malloc, this result is to be expected. The second potential cause of this result is an error within the technique. This would cause a reduced number of identifications when analysing the binary. It is logically implied that the cause of the error is in fact and error in porting as the Malloc Extension Patcher failed more when compared to Stackretencryption. This issue should be investigated further as it is difficult draw certain conclusions.

Chapter 7

Conclusions

7.1 Summary

As the cyber space grows in size and complexity, the number of sophisticated cyber attacks increase with it. Large organisations such as UQ are at the forefront of risk with extensive amounts of user data, resources and sensitive information. With the future of UQ in mind, this thesis proposed and provided a proof of concept for an automated UQ defence system. An aforementioned defence system would equip UQ to defend against cyber attacks at machine scale speed.

This project achieved this proof of concept by collating a collection of deprecated prior art designed for the Cyber Grand Challenge. As well as it, provided information and demonstration around the initial environment setup, porting and the use of Patcherex, an existing automatic patching tool. The procedures outlined in Section 4 and Section 5.2 allow future students or researchers to progress development in a structured position. Furthermore, the results discussed in Section 6.1 exhibited the success of demonstrating the use of Patcherex by inserting a custom code snippet into a binary file and modifying its behaviour. But, also outlined that the entire process was not generalised enough. Resulting in over fitting such that the results in Section 6.1 were undesirable. With only two successful insertions of found patches into target binaries, this highlights that the porting process outlined in Section 5.1 requires additional development.

7.2 Future Work

The objective of this project was not to create a complete product, but to instantiate and conceptualise a framework and proof of concept that can be built upon. Additionally, the sub optimal aforementioned results promote supplementary development of the methods and tools used in this thesis. Meaning, there is a lot of further work that is required before UQ can claim and rely on a fully autonomous defence system.

7.2.1 Further porting

Several parts of Patcherex were ported in this thesis to demonstrate functionality. However, this was only a fraction of the entire Patcherex tool. Leaving many components of Patcherex unusable until they are ported as well. The first of which is the remaining wrappers for detection techniques. It is recommended that a method wrapper is written for all techniques, allowing unrestricted use when calling patch_master. A template to follow was provided in Section 5.2.1. Secondly, although it was out of the scope of this project, it is proposed that further time is spent in porting the Reassembler Backend.

Reassembler Backend

Since it was originally designed for DECREE this component implements a custom decompiler designed for specific binaries. For use in a familiar system such as Linux, future developers must find a solution to decompilation. For this, there is the potential for integration of the complex SRE tool Ghidra introduced in Section 3.2. Or, applying existing decompilation techniques [**?**]. Porting the Reassembler Backend will further equip UQ's custom self defence system with methodologies to insert patches into vulnerable binaries. It should be noted that this technique will require termination of the running binary and re running of the patched binary. This will result in down time if the target binary is providing an online service. To mitigate this, an additional linking tool will need to be developed. This tool should aim to direct function execution from the original executable to the patched executable

without terminating the original program.

7.2.2 New POV framework

The CGC's POV solution details an exploit in a generalised and universal format. This allows quick testing of potential exploits against target binaries. UQ should mimic this methodology by structuring a custom built testing and POV framework, building upon the work completed in this thesis project (see Section 5.3). This supports modular testing, and encourages further development into other areas such as automated exploitation. Where the autonomous exploitation tool outputs a list of POVs to verify its success.

7.2.3 System monitoring

Finally, this project demonstrated the use of automatic tools, but still required human triggers to execute these tools. A fully autonomous system would require an external monitoring system which triggers Patcherex runs based on certain events. In an ideal scenario, an autonomous exploitation tool would trigger these events after discovering an exploit. It would then provide a POV to the automatic patching framework which would attempt to patch the vulnerability. However, there are many ways to approach this problem, and it is recommended to explore several paths.

7.2.4 Contribution

If desired, please contact me for transfer of ssh keys, allowing access to the zone. This will promote much needed future development of the project.

Appendix A

Decree System Calls

Name	Functionality
Terminate	*The equivalent of exit() which terminates the process*
Transmit	*The equivalent of send() which transmits a message to another socker.*
Receive	*The equivalent of recv() which reads data from a socket.*
Fdwait	*The equivalent of select() which examines the status of open input and output channels.*
Allocate	*The equivalent of mmap() which maps files or devices into memory*
Deallocate	*The equivalent of munmap() which unmaps files or devices into memory*
Random	*A system call that would generate random data*

Appendix B

General Purpose Registers

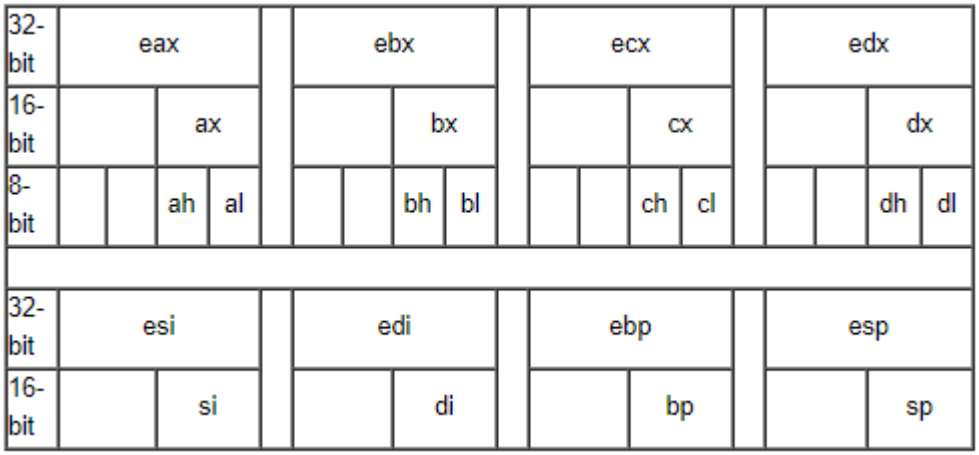

Figure B.1: General purpose registers structure

Appendix C

UQ Zone Specifications

This architecture refers to the z1-standard triton zone [50].

C.0.1 CPU usage

The percentages below are percentiles of a single CPU core. CPU cap is the maxmimum CPU resources the zone can consume at any given time. Guaranteed CPU describes the average CPU time that the zone can rely on. Even in worst case scenarios where the server is full and zones are competing for resources. Finally, burst ratio is the ratio between those two quantities.

- Guaranteed CPU 3%
- CPU cap 100%
- Burst ratio 30x

C.0.2 Operating system

Ubuntu 16.04 LTS

C.0.3 Network

UQ zone/uqcloud.net

Appendix D

Discussion Calculations

D.0.1 Failed during analysis

Of all of the results which failed during the analysis phase. ie. failed execution without patches from 6.1 (63). The percentages relating to which technique caused the failure. So, 38% failed due to Stackretencryption and 62% failed due to the Malloc Extension Patcher.

D.0.2 Successfully found patches

The Stackretencryption technique found patches 64% of the time and the Malloc Extension Patcher only found patches 21% of times. These values are not meant to correlate to 100% as it is a success percentile of each method.

D.0.3 Failed due to memory bus

All failed runs were due to memory bus error.

Bibliography

[1] 2007. [Online]. Available: https://eng.umd.edu/news/story/study-hackers-attack-every-39-seconds

[2] [Online]. Available: https://www.ibm.com/security/data-breach

[3] D. Fraze, "Cyber grand challenge," 2016. [Online]. Available: https://www.darpa.mil/program/cyber-grand-challenge

[4] "Patcherex," 2016. [Online]. Available: https://github.com/angr/patcherex

[5] K. Vorobyov, N. Kosmatov, and J. Signoles, "Detection of security vulnerabilities in c code using runtime verification: An experience report," in *Tests and Proofs*, C. Dubois and B. Wolff, Eds. Springer International Publishing, Conference Proceedings, pp. 139–156.

[6] H. C. A. van Tilborg and S. Jajodia, Eds., *Static Code Analysis*. Boston, MA: Springer US, 2011, pp. 1256–1256. [Online]. Available: https://doi.org/10.1007/978-1-4419-5906-5$_1$371

[7] B. Chess and G. McGraw, "Static analysis for security," *IEEE Security Privacy*, vol. 2, no. 6, pp. 76–79, Nov 2004.

[8] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv.org*, 2018.

[9] H. Li, H. Kwon, J. Kwon, and H. Lee, "Clorifi: software vulnerability discovery using code clone verification," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 6, pp. 1900–1917, 2016.

[10] A. Russo and A. Sabelfeld, "Dynamic vs. static flow-sensitive security analysis," in *2010 23rd IEEE Computer Security Foundations Symposium*, Conference Proceedings, pp. 186–199.

[11] F. Gao, L. Wang, and X. Li, "BovInspector: Automatic inspection and repair of buffer overflow vulnerabilities," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2016, pp. 786–791.

[12] M. Mohammadi, *Automatic Unit Testing to Detect Security Vulnerabilities in Web Applications*. ProQuest Dissertations Publishing, 2018. [Online]. Available: http://search.proquest.com/docview/2139138814/

[13] J. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles*, ser. SOSP '09. ACM, 2009, pp. 87–102.

[14] S. Cesare and Y. Xiang, *Dynamic Analysis*. London: Springer London, 2012, pp. 51–56. [Online]. Available: https://doi.org/10.1007/978-1-4471-2909-7_6

[15] K. A. Roundy and B. P. Miller, "Hybrid analysis and control of malware," in *Recent Advances in Intrusion Detection: 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, vol. 6307, pp. 317–338.

[16] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016.

[17] J. Bowring, A. Orso, and M. J. Harrold, "Monitoring deployed software using software tomography," in *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Conference Proceedings, pp. 2–9.

[18] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, "Measuring, analyzing and predicting security vulnerabilities in software systems," *computers security*, vol. 26, no. 3, pp. 219–228, 2007.

[19] M. L. Saboff, "Memory management techniques for on-line replaceable software," mar 6 2001, uS Patent 6,199,203.

[20] G. Hunt and D. Brubacher, "Detours: Binaryinterception ofwin 3 2 functions," in *3rd usenix windows nt symposium*, Conference Proceedings.

[21] C. Cifuentes and K. J. Gough, "Decompilation of binary programs," *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.

[22] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Conference Proceedings, pp. 353–368.

[23] K. H. Bennett and V. T. Rajlich, "Software maintenance and evolution: a roadmap," in *Proceedings of the Conference on the Future of Software Engineering*, Conference Proceedings, pp. 73–87.

[24] G. Candea, S. Bucur, and C. Zamfir, "Automated software testing as a service," in *Proceedings of the 1st ACM symposium on Cloud computing*, Conference Proceedings, pp. 155–160.

[25] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Conference Proceedings, pp. 2139–2154.

[26] T. Shellphish, "Cyber grand shellphish," *Phrack Papers*, 2017. [Online]. Available: https://www.yancomm.net/papers/2017%20-%20Phrack%20-%20Cyber%20Grand%20Shellphish.pdf

[27] "Decree," 2016. [Online]. Available: https://archive.darpa.mil/cybergrandchallenge/tech.html

[28] M. N. Hoda, N. Chauhan, S. M. K. Qua, and P. R. Srivastava, *Software engineering : proceedings of CSI 2015.* Singapore: Springer, 2019.

[29] "Angr." [Online]. Available: https://angr.io/

[30] G. F. Elsayed, I. Goodfellow, and J. Sohl-Dickstein, "Adversarial reprogramming of neural networks," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=Syx$_S$s05tm

[31] M. F. Thompson, "Effects of a honeypot on the cyber grand challenge final event," *IEEE Security Privacy*, vol. 16, no. 2, pp. 37–41, 2018.

[32] "Fidget," 2019. [Online]. Available: https://github.com/angr/fidget

[33] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking.* Springer, 2018, pp. 305–343.

[34] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2008, pp. 337–340.

[35] 2019. [Online]. Available: https://github.com/Z3Prover/z3

[36] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP).* IEEE, 2019, pp. 985–999.

[37] B. Niu and G. Tan, "Modular control-flow integrity," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.

[38] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, 2018, pp. 81–116.

[39] I. Z. Avraham, "Non-executable stack arm exploitation research paper," *Revision*, vol. 1, pp. 2010–2011, 2010.

[40] D. Bartholomew, "Qemu: a multihost, multitarget emulator," *Linux Journal*, vol. 2006, no. 145, p. 3, 2006.

[41] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, vol. 41, 2005, p. 46.

[42] "Gcc, the gnu compiler collection," 2020. [Online]. Available: https://gcc.gnu.org/

[43] "Cb multios," 2016. [Online]. Available: https://github.com/trailofbits/cb-multios

[44] R. M. Stallman, *Using and porting GNU CC*. Free Software Foundation, 1998.

[45] "Ghidra," 2020. [Online]. Available: https://ghidra-sre.org/

[46] "Registers," 2012. [Online]. Available: https://wiki.skullsecurity.org/index.php?title=Registersesi

[47] "x86 and amd64 instruction reference," 2019. [Online]. Available: https://www.felixcloutier.com/x86/

[48] "Push all general-purpose registers," 2020. [Online]. Available: http://qcd.phys.cmu.edu/QCDcluster/intel/vtune/reference/vc267.htm

[49] "Linux servers," 2016. [Online]. Available: https://web.archive.org/web/20160812230329/http://www.w3cook.com:80/os/summary

[50] "Uqcloud zones guide," 2013. [Online]. Available: https://stluc.manta.uqcloud.net/xlex/public/zones-guide.html

[51] "Debugging with gdb," 2020. [Online]. Available: https://wiki.python.org/moin/DebuggingWithGdb

[52] "The gnu project debugger," 2020. [Online]. Available: https://www.gnu.org/software/gdb/

[53] "The python debugger," 2020. [Online]. Available: https://docs.python.org/3/library/pdb.html

[54] "Povsim," 2016. [Online]. Available: https://github.com/mechaphish/povsim