MSc Data Science dissertation

Dundee University

EXPLOITING ROP CHAIN VULNERABILITIES WITH REINFORCEMENT LEARNING

Marieke Gijsberts, 2415540

Januari 2022

Abstract

This project aimed to use reinforcement learning to find ROP chain vulnerabilities, a type of cybersecurity exploit that can lead to arbitrary code execution. As a security researcher, we would want to find such vulnerabilities in a program and warn our client, before attackers can use the vulnerability to cause damage.

To be able to exploit a ROP chain vulnerability an attacker first needs a way to subvert the program's control flow from its normal course, usually through a buffer overflow, after which they can try to redirect the program's execution to perform malicious activity. The buffer overflow would be the main issue in this case, but actually exploiting it would prove to the client just how bad it actually is. Finding a ROP chain can be tedious work however, which is why I wanted to automate it. To do this I used Q-learning to try and solve ROP chain CTF challenges, a gamified version of an actual ROP chain vulnerability.

While working on this project I realised that currently there appears to be no method available to execute the project in the way I really wanted. A ROP chain is usually made up of several steps (called 'gadgets'), and I wanted to use each gadget as a separate action and evaluate the effects of each of them separately. In a standard reinforcement learning implementation, an agent takes one action, gets feedback from that action and uses that to determine the next action it wants to take. When exploiting ROP chain vulnerabilities however, you only get one chance to provide your input to the application, meaning you have to supply a sequence of actions in one go. I could not find any current ways to implement this, but I'll theorise on how this could be done.

For my actual project I ended up with a more simplified setup, where I regard each possible sequence of gadgets as a single action. This proved to work quite well, especially when I was able to cater my rewards to the optimal gadget length. When I didn't adapt my rewards, the agent often converged to a correct but less optimal solution.

Student declaration

I declare that the special study described in this dissertation has been carried out and the dissertation composed by me, and that the dissertation has not been accepted in fulfilment of the requirements of any other degree or professional qualification.

Marieke Gijsberts

Supervisor certification

I certify that Marieke Gijsberts has satisfied the conditions of the Ordinance and Regulations and is qualified to submit this dissertation in application for the degree of Master of Science.

Karen Petrie

Contents

Introduction **8**

1 Background **10**

1.1 The CPU, instructions and registers 10

1.2 The stack and stack buffer overflows 11

1.3 ROP: Return-oriented programming 13

2 Solving a ROP chain challenge **15**

2.1 Thoughts on 'translating' this manual process to an automatic one 19

3 Reinforcement learning **21**

3.1 Why reinforcement learning? 21

3.2 Important components of reinforcement learning 22

3.2.1 Exploration vs exploitation 22

3.2.2 Rewards & values 22

3.2.3 State, policy and model 23

3.3 K-armed bandits 24

3.4 Q-learning 24

4 Project: setup **27**

5 Project: automating the exploit attempts **28**

5.1 When do we want the register values? 28

5.2 Pwntools & gdb 28

5.3 Which gadgets to provide to the RL agent? 29

6 Project: automating the reinforcement learning **30**

7 Project: experimenting and analysing **32**

8 Project: results **34**

8.1 Setup (a) 34

8.2 Setup (b) 36

8.3 Setup (c) . . . 37
8.4 Setup (d) . . . 38
8.5 Project: conclusions . . . 39

9 Conclusions **40**

10 Discussion **42**
10.1 Limitations of this project . . . 42
10.2 Convergence . . . 42

11 Suggestions for further research **44**
11.1 Gadgets . . . 44
11.2 Hybrid learning . . . 44
11.2.1 Other hybrid learning areas . . . 44
11.2.2 Deciding on the sequence length . . . 45
11.2.3 Deciding on the sequence . . . 46

12 References **49**

A Original planning **51**

B Results **52**
B.1 Exploration vs Exploitation . . . 52
B.2 Pick percentages . . . 53

List of Figures

1 x86 ISA base register set (Crutcher et al., 2021) . . . 11
2 Stack buffer overflow (Goedegebure, 2020) . . . 12
3 Program layouts (Roemer et al., 2012) . . . 13
4 Split, an application vulnerable to ROP chain attacks . . . 15
5 Strings present in the Split binary . . . 15
6 The address of "pop rdi ; ret" . . . 16
7 A call to the system() function . . . 16
8 The pwnme() function . . . 17
9 Creating a pattern . . . 17
10 Finding the offset . . . 18
11 The full code for our exploit . . . 19
12 Q-learning process (Chang et al., 2019) . . . 26
13 Setup (a): exploration vs exploitation per 100 episodes, in percentages . . . 34
14 Setup (a): results per 100 episodes, in percentages . . . 35
15 Setup (b): exploration vs exploitation per 100 episodes, in percentages . . . 36
16 Setup (b): results per 100 episodes, in percentages . . . 36
17 Setup (c): exploration vs exploitation per 100 episodes, in percentages . . . 37
18 Setup (c): results per 100 episodes, in percentages . . . 37
19 Setup (d): results per 100 episodes, in percentages . . . 38
20 Monster Train (Northernlion, 2020) . . . 45
21 Hybrid learning process . . . 48
22 Original planning for the project . . . 51
23 B.1 (a) . . . 52
24 B.1 (b) . . . 52
25 B.1 (c) . . . 52
26 B.1 (d) . . . 53
27 B.2 (a) . . . 53
28 B.2 (b) . . . 53
29 B.2 (c) . . . 53

30 B.2 (d) 54

List of Tables

1 Possible solves - simplified 31
2 Different experiment setups 32
3 Successful solves as percentage of total number of permutations 46

Introduction

The primary goal of this project is to solve ROP chain exploitation challenges using reinforcement learning. ROP stands for return-oriented programming, a type of cybersecurity exploit that I will explain further in paragraph 1.3. I will attempt to solve ROP chain challenges using Q-learning (which I will talk about in paragraph 3.4), after which I will discuss possible approaches that might lead to a better result for this specific reinforcement learning problem.

In this project I will attempt to solve ROP chain capture the flag (CTF) challenges. CTFs are a special kind of information security competitions. These competitions usually have several tasks in a wide range of security categories, such as web, forensic, cryptography and binary exploitation (CTFtime team, n.d.). For this project I've collected several ROP chain challenges, which are freely available online.

In these challenges it is usually the goal to find a specific text know as a flag, but in the real world ROP chain vulnerabilities can lead to any sort of malicious activity. ROP chain vulnerabilities often start with an insecure input, which an attacker can exploit if there are enough 'gadgets' available in the application. Automatically searching for these combinations of gadgets seems like a good way to speed up the process of vulnerability research: once a researcher finds a vulnerable input that should be enough information to tell their client that their application is vulnerable, but being able to show just how wrong things could go might lead to bigger impact. This can be tedious work however, which is why I sought to automate it by training a reinforcement learning model.

While trying to solve these challenges using reinforcement learning, I ran into a specific contradiction: with most reinforcement learning strategies, the agent takes one action at a time and then uses the feedback it receives to determine the next action. When solving ROP chain challenges however, it is usually only possible to send in a sequence of actions in one go. I was not able to overcome this contradiction completely, which is why I had to resort to a less extensive learning setup, but I will theorise on how this

could be achieved.

But before I continue about the actual project, please allow me to give some general background information on what ROP chain vulnerabilities are and how they can be found. I will also give a brief overview on reinforcement learning, and then take a look at how I combined the two for this project.

1 Background

1.1 The CPU, instructions and registers

It's the job of the CPU (central processing unit) in a computer to fetch, decode, execute, and store the results of instructions (Crutcher et al., 2021). Instructions are defined per a specification, called instruction set architecture, or ISA. the instruction set will have instructions for doing arithmetic, moving data between memory locations (registers or main memory), controlling the flow of execution, and more (Crutcher et al., 2021). Many different instruction sets exist, for this project we will look at applications that use the x86_64 ISA, the 64 bits version of x86.

Crutcher et al. (2021) explain that CPUs have special memory locations called registers. Registers are used to store values in the CPU that help it execute instructions without having to refer back to main memory. The CPU will also store results of operations in registers. This enables you to instruct the CPU to do calculations between registers and avoid excess memory accesses (Crutcher et al., 2021). Figure 1 shows the x86_64 registers.
In this project I intended to use the values in these registers as the feedback for the reinforcement learning agent, to show it what the effect of an action was.

x86_64 assembly code uses 64-bit registers. Additionally, the lower bytes of some of these registers may be accessed independently as 32-, 16- or 8-bit registers (Doeppner, 2019). Most registers can be used interchangeably, with some exceptions: rsp is the stack pointer, which can be manipulated by instructions such as push and pop; rbp is conventionally the frame pointer, as reflected in instructions like enter and leave; rsi and rdi are the source and destination registers for certain string operations (Roemer et al., 2012).

Table 1-1. *x86 Base Register Set*

	64 bits (x86_64)	32 bits (x86)	16 bits(8086) 8 bits	8 bits
Accumulator	RAX	EAX	AX	
			AH	AL
Base register	RBX	EBX	BX	
			BH	BL
Counter	RCX	ECX	CX	
			CH	CL
Data	RDX	EDX	DX	
			DH	DL
Base pointer	RBP	EBP	BP	
				BPL
Source index	RSI	ESI	SI	
				SIL
Destination index	RDI	EDI	DI	
				DIL
Stack pointer	RSP	ESP	SP	
				SPL
General purpose	R8-R15	R8D-R15D	R8W-R15W	
				R8B-R15B

Figure 1: x86 ISA base register set (Crutcher et al., 2021)

1.2 The stack and stack buffer overflows

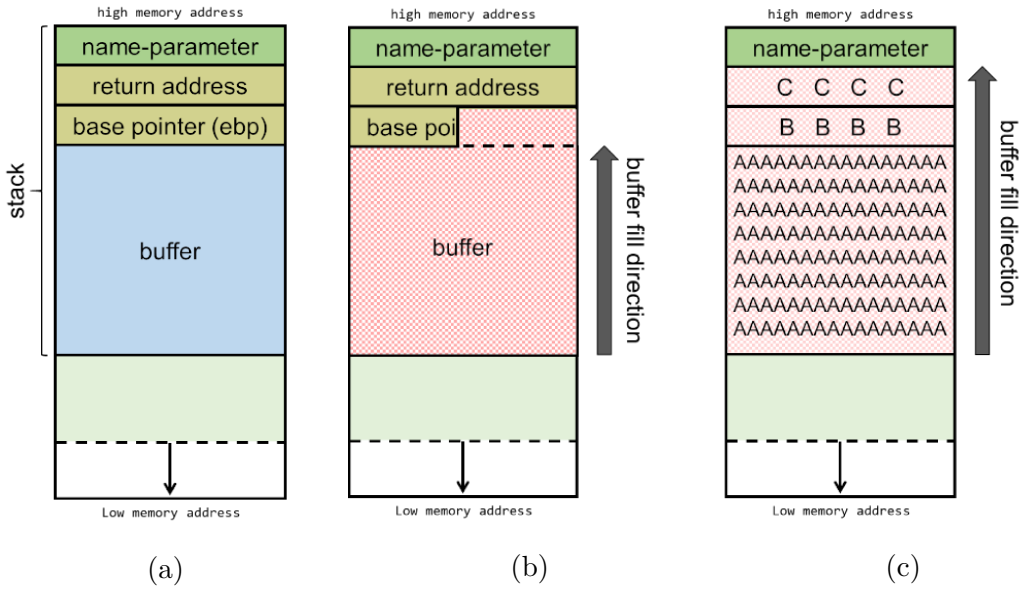
One (1996) state that a stack is an abstract data type frequently used in computer science. A stack of objects has the property that the last object placed on the stack will be the first object removed. This property is commonly referred to as last in, first out queue, or a LIFO (One, 1996).

The CPU uses the stack – a contiguous block of memory (One, 1996) – to keep track of function parameters, local variables, and return addresses (Crutcher et al., 2021).

For the x86_64 ISA, there are two instructions to work with the stack: push and pop (Crutcher et al., 2021). Push adds an element at the top of the stack. pop, in contrast, reduces the stack size by one by removing the last element at the top of the stack (One, 1996).

Crutcher et al. (2021) explain that there's also a special register called the extended stack pointer (rsp). The x86_64 stack always starts at a high memory address. As data is pushed onto the stack, the rsp decrements to the next address. When the pop instruction is executed, the rsp increments to reveal the previous item on the stack. We use the rbp register, also called the base pointer, to save the value of rsp before we

change it (Crutcher et al., 2021).



(a) (b) (c)

Figure 2: Stack buffer overflow (Goedegebure, 2020)

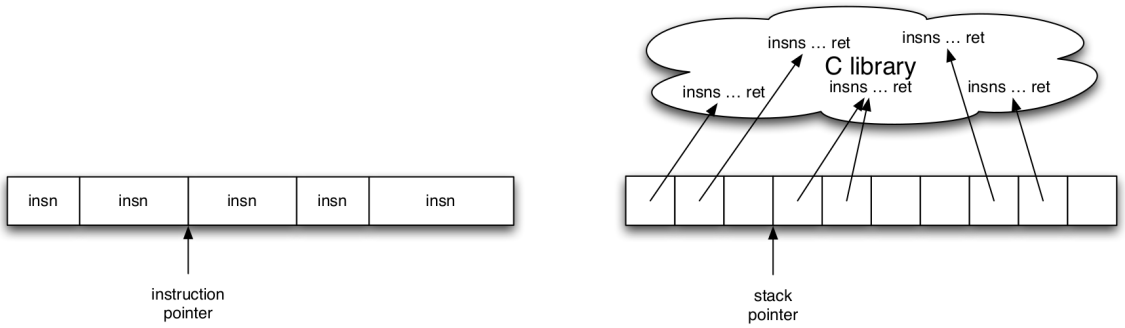
A buffer overflow is the result of stuffing more data into a buffer than it can handle (One, 1996). Memory overflows are one of the topmost causes of vulnerabilities (Crutcher et al., 2021). Gerg (2005) states that buffer overflows have been detected in many types of software ranging from web browsers to web servers. Software such as Internet Explorer, PHP and Apache all have been victim to such vulnerabilities (Gerg, 2005). Languages that expect the programmer to explicitly manage memory (e.g., C, C++) are more vulnerable to these kind of attacks than programming languages that provide automatic memory management and garbage collection (e.g., C#, Java, Python) (Crutcher et al., 2021).

In languages that need explicit memory management, the developer has to define the length of each buffer it wants to reserve on the stack, and then make sure this buffer gets respected. This is however not guaranteed: the standard C library for instance provides a number of functions for copying or appending strings, that perform no boundary checking (One, 1996). This means that using any one of these functions could lead to a buffer overflow.

Figure 2a shows the 'normal' state of the stack, Figure 2b shows how a vulnerable function makes it possible to overwrite past the set buffer, causing a stack buffer overflow, Figure 2c shows how the return address gets overwritten, making it possible to control what memory address the application will jump to next.

1.3 ROP: Return-oriented programming

Return-oriented programming is a technique introduced by Roemer et al. (2012) by which an attacker can induce arbitrary behavior in a program whose control flow he has diverted — without injecting any code. A return-oriented program chains together short instruction sequences already present in a program's address space, each of which ends in a "return" instruction, as illustrated in Figure 3 (Roemer et al., 2012).



(a) Layout of an ordinary program

(b) Layout of a return-oriented program

Figure 3: Program layouts (Roemer et al., 2012)

ROP is a way to take the code the programmer used to make the program do one thing, and then using it to do something completely different.

According to Roemer et al. (2012), to use a ROP chain to attack an application the attacker must (1) subvert the program's control flow from its normal course, and (2) redirect the program's execution. The most familiar way to subvert the programs control flow is to use a stack buffer overflow, but other vulnerabilities might work as well. (Roemer et al., 2012)

Roemer et al. (2012) states that a usual target for ROP chain attacks is the standard C library, libc, since it is loaded in nearly every Unix program and contains routines of the sort that are useful for an attacker (e.g., wrappers for system calls). Such attacks are therefore known as return-into-libc attacks. However, in principle any available code, either from the program's 'own' code or from a library to which it links, could be used (Roemer et al., 2012).

As shown in Figure 3a, a standard program contains of several instructions that get 'visited' one at a time in (mostly) chronological order (of course there can be loops and jumps) as the instruction pointer progresses through the program. A return-oriented program however, as shown in Figure 3b, instead jumps from address on the stack to address on the stack. At each address it jumps to, it runs the code from that address until it gets sent back to the stack by a ret command, thereby completely changing the order in which lines of code get executed from the way it was originally intended.

2 Solving a ROP chain challenge

In this chapter, I will show how ROP chain exploits are found by solving one of the ROP Emporium challenges by hand.
I will demonstrate how to solve the ROP Emporium challenge Split (ROP Emporium, n.d.). This is an application that can be run from the command line, and expects one single input. After the user has given some input it says Thank you! and then exits, as shown in Figure 4.

```
split by ROP Emporium
x86_64

Contriving a reason to ask user for data...
> test
Thank you!

Exiting
```

Figure 4: Split, an application vulnerable to ROP chain attacks

The description for this challenge says that the useful string "/bin/cat flag.txt" is present somewhere in this binary. Executing this code will return the contents of the file flag.txt. Therefore my first step is to analyse all strings that the developer put in this binary, as shown in Figure 5. In the second column we can see the addresses of the strings we found. As we can see the last entry in this list refers to /bin/cat flag.txt. Now we know one aspect of our ROP chain: the address 0x601060.

```
[Strings]
nth paddr      vaddr      len size section type  string
-------------------------------------------------------
0   0x000007e8 0x004007e8 21  22   .rodata ascii split by ROP Emporium
1   0x000007fe 0x004007fe 7   8    .rodata ascii x86_64\n
2   0x00000806 0x00400806 8   9    .rodata ascii \nExiting
3   0x00000810 0x00400810 43  44   .rodata ascii Contriving a reason to ask user
 for data...
4   0x0000083f 0x0040083f 10  11   .rodata ascii Thank you!
5   0x0000084a 0x0040084a 7   8    .rodata ascii /bin/ls
0   0x00001060 0x00601060 17  18   .data   ascii /bin/cat flag.txt
```

Figure 5: Strings present in the Split binary

To get our vulnerable application to run this bit of code, we will need to add it to the rdi register, since in x86_64 the registers rdi, rsi, rdx, rcx, r8, and r9 are used to pass the first six integers or pointer parameters to called functions (Doeppner, 2019). This means that if we can place our string in the rdi register and then call system(), the system()

function will then use the string /bin/cat flag.txt as its first (and only) input parameter, which will give us the contents of flag.txt.
Searching for a gadget that pops the top of the stack into the rdi register gives us the result as shown in Figure 6, at address 0x4007c3.

```
mples/ROPEmporium/rop_emporium_all_challenges/split$ ROPgadget --binary ./split
--ropchain | grep "pop rdi ; ret"
0x00000000004007c3 : pop rdi ; ret
```

Figure 6: The address of "pop rdi ; ret"

Now we will need to know where we can find the line of code that calls the system() function. For this I opened the file in Ghidra, a software reverse engineering tool developed by the NSA (National Security Agency, 2021). As shown in Figure 7, the binary contains a function called UsefullFunction, which contains a CALL to system at address 0x40074b.

```
                    undefined usefulFunction()
     undefined        AL:1          <RETURN>
                    usefulFunction                                  XREF[2]:
00400742 55             PUSH        RBP
00400743 48 89 e5       MOV         RBP,RSP
00400746 bf 4a 08       MOV         EDI=>s_/bin/ls_0040084a,s_/bin/ls_0040084a
         40 00
0040074b e8 10 fe       CALL        <EXTERNAL>::system
         ff ff
00400750 90             NOP
00400751 5d             POP         RBP
00400752 c3             RET
```

Figure 7: A call to the system() function

Now that we know which addresses to hop to to execute our code, we will need to figure out how to exploit our vulnerable input. Looking at the code in Ghidra, we can see that the main function calls a function called pwnme(), which reserves a buffer in memory of 0x20 length, but then accepts an input of 0x60 length as shown in Figure 8.
This means that this input will accept a longer input than it has reserved memory space for, making it possible for us to write directly to the stack after filling up the expected buffer. We can use gdb to find out how long this initial input needs to be, as shown in Figure 9 and Figure 10.

```
void pwnme(void)

{
  undefined local_28 [32];

  memset(local_28,0,0x20);
  puts("Contriving a reason to ask user for data...");
  printf("> ");
  read(0,local_28,0x60);
  puts("Thank you!");
  return;
}
```

Figure 8: The pwnme() function

```
gef➤  pattern create 60
[+] Generating a pattern of 60 bytes (n=4)
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaa
[+] Saved as '$_gef0'
gef➤
```

Figure 9: Creating a pattern

In Figure 9 we create a De Bruijn sequence of 60 bytes, which should be more than long enough, and then we run the Split application again with that sequence as our input. This results in a Segmentation fault, since our input was way too long. We can now search for the pattern offset from the rsp register, which is the first register that got overwritten with our too long input. Figure 10 shows that the current contents of rsp can be found at offset 40, which means that after 40 bytes of input, we have overwritten the reserved space on the stack and are able to write the addresses we want to travel to for our exploit to the stack.

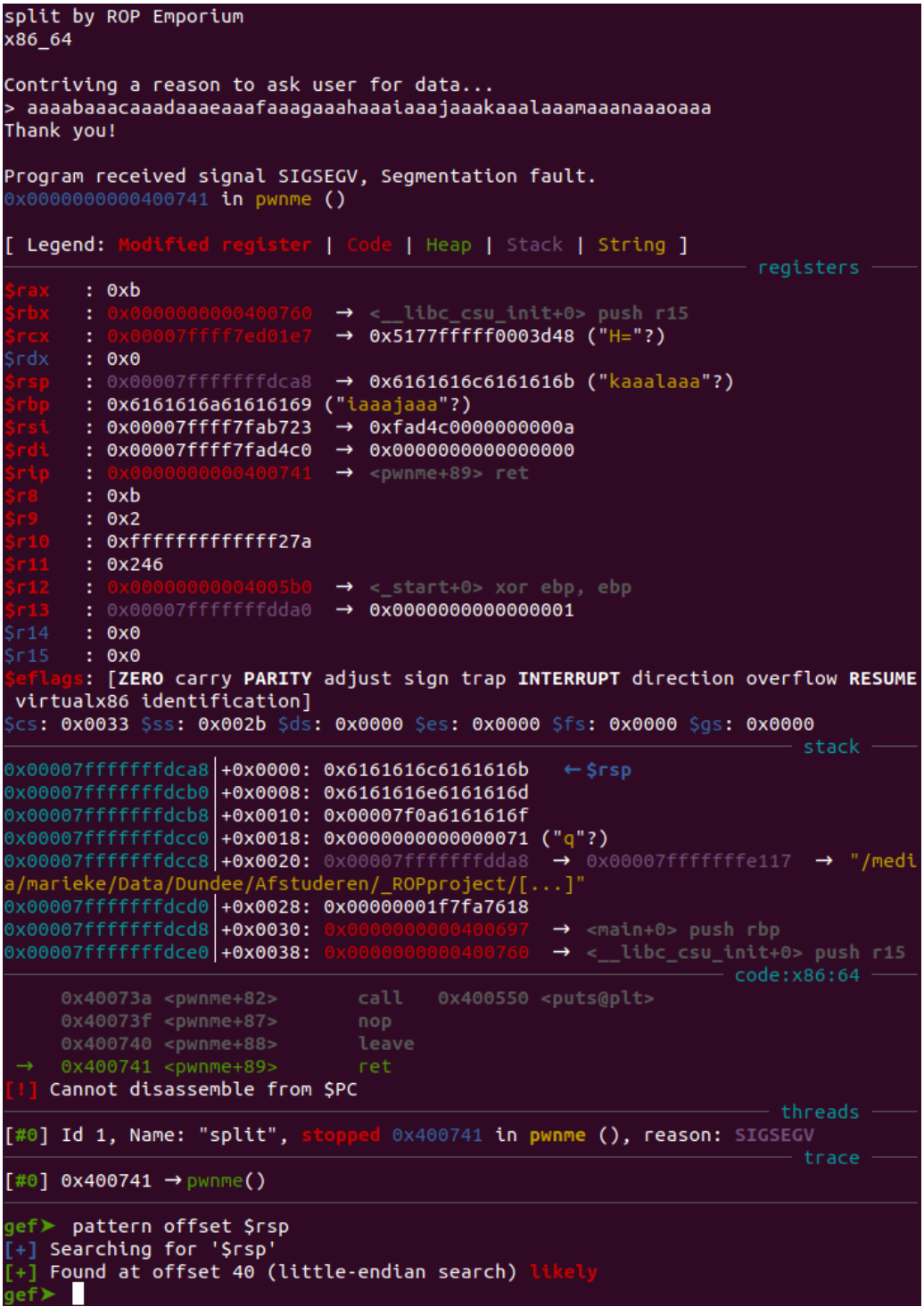


Figure 10: Finding the offset

Now we have everything we need to write our exploit, as shown in Figure 11. We can use pwntools (pwn) to create a connection to the binary. Then we create our payload: 40 bytes of buffer (in this case the character A) and then the three addresses we found

earlier. We send the payload, receive the response, use regex to search for a flag in the right format and then print that response. The code for this manual exploit can be found on Github.

```
#! /usr/bin/env python3
# coding: utf-8

import os
os.environ['PWNLIB_NOTERM'] = '1'
from pwn import *

#connect to the binary
exe = context.binary = ELF('./split')
io = process([exe.path])

# create payload
payload = b"A"*40 # start with buffer
payload += p64(0x4007c3) # pop rdi
payload += p64(0x601060) # /bin/cat flag.txt
payload += p64(0x40074b) # system()

# send payload
io.sendline(payload)

# receive response
response = io.recvall()
print(re.search("(ROPE{.*?})", response.decode()))
```

Figure 11: The full code for our exploit

2.1 Thoughts on 'translating' this manual process to an automatic one

Tools like ROPgadget can be used to find useful gadgets, but even though it found a total of 93 possible gadgets, it didn't find all gadgets we needed for this exploit. Tools like Ghidra can be useful to find other potentially useful addresses, but clearly this requires extra manual input from the researcher.

Another option would be to consider each line of code that precedes a RET statement to be a possible gadget. This would lead to a very long list of possible addresses as input for the reinforcement learning agent.

However sometimes a ROP chain exploit doesn't just exist of a list of memory addresses. For instance in the ROP Emporium challenge 'callme', the researcher also has to submit

several integers as input parameters to certain exploitable functions.

This would mean that most ROP chain exploits would require some initial reverse engineering and selection of useful addresses by the researcher, which can then be supplemented by 'standard' possibly useful gadgets as found by a tool like ROPgadget. This initial reverse engineering requirement does make reinforcement learning seem like a less ideal solution: my initial goal was to fully automate the steps after finding a stack buffer overflow vulnerability. Having to manually search for reasonable inputs first does mean that the productivity improvement over 'just doing it all by hand' becomes smaller.

When I decided to try to train a reinforcement learning model to solve ROP chain challenges, I was curious if it would pick up on the mechanisms needed to solve these challenges: a lot of times the challenge is to put some data (such as the address of the /bin/cat flag.txt in the above example) in specific registers, and then go to an address that processes these registers. This is why I wanted to use the contents of the registers as the state of the environment. I've written code that extracts the content of the registers after each step, but then I had to switch to a version of reinforcement learning where there is only one 'big' step, after which solving the challenge either succeeded or failed, and the contents of the registers don't get evaluated in that process. I was very interested to see if/how well the training would generalize (the specific 'good' register values would change from challenge to challenge) but sadly I didn't get to experiment with that.

3 Reinforcement learning

Reinforcement learning is learning what to do – how to map situations to actions – so as to maximize a numerical reward signal. The learner (called the agent) is not told which actions to take, but instead must discover which actions yield the most reward by trying them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards. These two characteristics – trial-and-error search and delayed reward – are the two most important distinguishing features of reinforcement learning (Sutton and Barto, 2018).

3.1 Why reinforcement learning?

The methods in machine learning can be divided into three main groups: supervised learning, unsupervised learning and reinforcement learning, as explained by Braga-Neto (2020):

The objective in supervised learning is to predict Y given X. The target Y is always defined and available in the training data. There are two main types of supervised learning problems: classification, where Y represents a label, not a numeric value, and regression, where Y represents a numerical quantity, which could be continuous or discrete (Braga-Neto, 2020).

In unsupervised learning, Y is unavailable and only the distribution of X specifies the problem. Therefore, there is no prediction and no prediction error, and it is not straightforward to define a criterion of performance. Unsupervised learning methods are mainly concerned in detecting structure in the distribution of X. Examples include dimensionality reduction and clustering (Braga-Neto, 2020).

Finally there is reinforcement learning, which concerns decision making in continuous interaction with an environment, where the objective is to minimize a cost (or maximize a reward) over the long run (Braga-Neto, 2020).

The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that evaluates the actions taken rather than instructs by giving correct actions. Purely evaluative feedback indicates how good the action taken was, but not whether it was the best or the worst action possible. This is what creates the need for active exploration, for an explicit search for good behavior. Purely instructive feedback on the other hand, indicates the correct action to take, which is the basis of supervised learning (Sutton and Barto, 2018).

The goals of my project strongly align with reinforcement learning. Perhaps the current execution of this project also could have worked as a supervised learning problem, since I only used one correct 'label' as answer to my problem, but my initial goal, that I still hope to achieve one day, would consist of learning from several interactions with the environment and come up with the best sequence of actions based on those interactions.

3.2 Important components of reinforcement learning

Sutton and Barto (2018) mention several important aspects of a reinforcement learning setup, which I will describe in this paragraph.

3.2.1 Exploration vs exploitation

One of the challenges that arises in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future (Sutton and Barto, 2018).

3.2.2 Rewards & values

A reward signal defines the goal of a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number called the

reward. The agent's sole objective is to maximize the total reward it receives over the long run. The reward signal thus defines what the good and bad events for the agent are (Sutton and Barto, 2018).

Whereas the reward signal indicates what is good in an immediate sense, a value function specifies what is good in the long run. Roughly speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states (Sutton and Barto, 2018).

3.2.3 State, policy and model

The state can be considered a signal conveying to the agent some sense of 'how the environment is' at a particular time. A learning agent must be able to sense the state of its environment to some extent and must be able to take actions that affect the state. The agent also must have a goal or goals relating to the state of the environment (Sutton and Barto, 2018).

A policy defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states (Sutton and Barto, 2018).

A model is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for planning, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called model-based methods, as opposed to simpler model-free methods that are explicitly trial-and-error learners and viewed as almost the opposite of planning

(Sutton and Barto, 2018).

3.3 K-armed bandits

One possible problem that can be solved by reinforcement learning is the k-armed bandit problem, so named by analogy to a slot machine, or 'one-armed bandit', except that it has k levers instead of one. Each action selection is like a play of one of the slot machine's levers, and the rewards are the payoffs for hitting the jackpot. Through repeated action selections you are to maximize your winnings by concentrating your actions on the best levers. Another analogy is that of a doctor choosing between experimental treatments for a series of seriously ill patients. Each action is the selection of a treatment, and each reward is the survival or well-being of the patient (Sutton and Barto, 2018).

K-armed bandits can be considered reinforcement learning in a simplified setting, one that does not involve learning to act in more than one situation (Sutton and Barto, 2018). My current setup for my project could have been solved by a k-armed bandit method: I am only considering one challenge and the agent only has to take one action, after which the episode is done and a success/failure feedback is returned to the agent. However my eventual goal was to see each used gadget as a separate action, and to learn from the results of using each individual gadget. Therefore, I decided to use a more extensive method – Q-learning – in the hopes of possibly expanding my initial setup during the project.

3.4 Q-learning

Q-learning (C. J. C. H. Watkins, 1989) is a form of model-free reinforcement learning. It provides agents with the capability of learning to act optimally in some discrete, finite environment by experiencing the consequences of actions, without requiring them to build maps of the environment (C. J. Watkins and Dayan, 1992). The goal of this form of reinforcement learning is to estimate the optimal Q-function – where Q is a function of state and action (Brunskill, 2019) – which measures the expected cumulative utility of

each currently available decision, given that the decision maker will follow the optimal decision strategy in the future (Clifton and Laber, 2020).

Q-learning works by incrementally updating the expected values of actions in states. For every possible state, every possible action is assigned a value which is a function of both the immediate reward for taking that action and the expected reward in the future based on the new state that is the result of taking that action (Gaskett et al., 1999). These values all start at zero, and get updated after a state-action pair is executed.

This is expressed by the one-step Q-update equation:

$$Q(s,a) := (1-\alpha)Q(s,a) + \alpha(R + \gamma maxQ(s_{t+1}, a_{t+1}))$$

where Q is the expected value of performing action a in state s; s is the state vector; a is the action vector; R is the reward; α is a learning rate which controls convergence and γ is the discount factor, which makes rewards earned earlier in the process more valuable than those received later (Gaskett et al., 1999).

Q-learning is exploration insensitive, any action can be carried out at any time to gain new information from this experience. In the standard Q-learning implementation Q-values are stored in a table. One cell is required per combination of state and action. This implementation is not amenable to continuous state and action problems (Gaskett et al., 1999).

The process of Q-learning is explained in Figure 12, where ε stands for the exploration rate and the two options '1 - ε' and 'ε' refer to the ε-greedy algorithm. In this approach, the agent takes the current optimal action based on the current Q function with probability 1 - ε, where $\varepsilon \in [0, 1]$. With probability ε, the agent takes a random action. Thus, the agent balances exploration with the random actions and exploitation with the optimal actions. Larger ε promote more random exploration. Typically, the value of ε will be initialized to a large value, often (as I did in this project) as $\varepsilon = 1$. Throughout the course of training, ε decays so that as the Q function improves, the agent

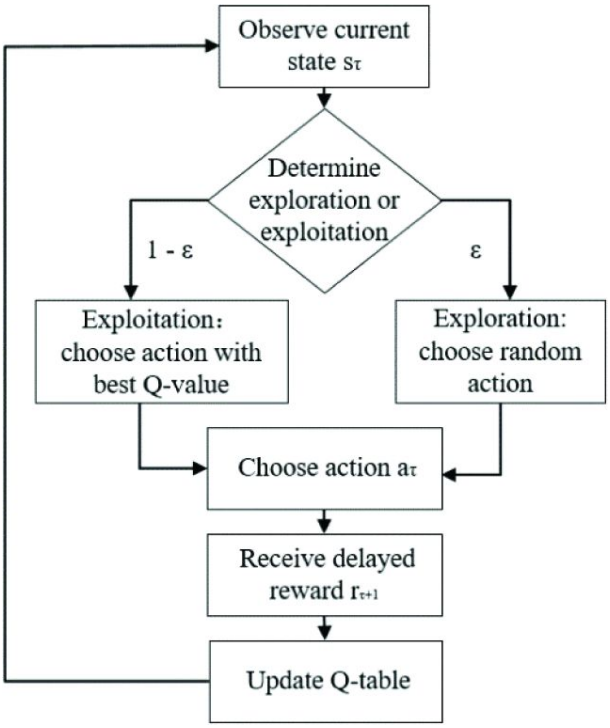


Figure 12: Q-learning process (Chang et al., 2019)

increasingly takes the current optimal action (Brunton and Kutz, 2021). In my project I've experimented with different exploration decay rates.

Q-learning is based on Markov decision processes. A Markov decision process consists of four parts: a state-space S (the set of possible states of the system to be controlled), a function A that gives the possible actions for each state, a transition function T and a reward function R. In each state, the controller of the system may perform any of a set of possible actions (C. J. C. H. Watkins, 1989). Any process can be modelled as a Markov process if the state-space is made detailed enough to ensure that a description of the current state captures those aspects of the world that are relevant to predicting state-transitions and rewards (C. J. C. H. Watkins, 1989).

4 Project: setup

For this project I trained a Q-learning model on the Split challenge from ROP emporium (ROP Emporium, n.d.). The project can be described in three steps: automating the ROP chain exploit process, automatic the reinforcement learning and analysing the results.

My original planning for this project can be found in appendix A. My original intent was to see how far I could get in automating the entire process. The first step would be to train a RL model to find the correct sequence of ROP gadgets to add to the buffer (as demonstrated in this project), the second step would have been to see if I could train a model on finding the correct initial buffer, and a third stretch goal was to train the entire process, which would have included finding the vulnerable input as well. In the end I purely focused on step 1, and I only used the CTF challenges as found on ROP Emporium.

The training was done on a desktop pc running Ubuntu 20.04, on 16 GB of DDR4 RAM and an AMD Ryzen 9 3900X processor. Tools used to run the ROP chain exploits were GNU gdb 9.2, Pwntools v4.6.0 and ROPgadget v6.6. Analysis was done using Numpy version 1.21.3, Pandas version 1.3.5, Matplotlib version 3.5.1 and Bokeh version 2.4.2.

5 Project: automating the exploit attempts

As stated earlier, my initial plan was to use the contents of the registers as feedback to the reinforcement learning agent. This meant that I had to interact with the vulnerable application from two 'sides': on the one hand I needed to be able to automatically supply the input (the chain of gadgets) and receive the output the application would give me (let's call this the 'front-end') on the other hand I needed to automatically obtain the contents of the registers (through the 'back-end' so to speak). In order for my RL agent to automatically interact with the vulnerable application many times in a row, I needed to get this front and back-end information using one Python script.

5.1 When do we want the register values?

As we've seen in chapter 2, jumping to one ROP gadget can lead to several lines of code getting executed, from the line of code we initially jump to until we get sent back to the stack through a RET command. If we consider the contents of the registers to show the effect of our gadgets, you would have to argue that we want to know their contents right before we run into that RET command that sends us back to the stack.
This turned out to be very hard to automate, plus there is always the possibility that a specific gadget doesn't end at a RET command – the final gadget in the 'correct' solution for the Split challenge for instance ends with returning the flag, after which the program ends and there are no more register values to request. This made me decide to start with obtaining the register values right after jumping to that specific address.
I am unsure if this would lead to usable enough information this way and so I'm still trying to think of different/better options for this (getting the register values on the line right after the line we jumped to might have been an option, assuming we'll never jump to the very last line of code in a program) but since I had to turn to a more simplified reinforcement learning implementation in the end, I didn't get to experiment on this.

5.2 Pwntools & gdb

Sending and receiving the front-end can be done by using Pwntools, as demonstrated in chapter 2. Obtaining the registers from the back-end can be done using gdb, a

debugging tool already shown in Figures 9 and 10. Initial research I did when designing my project proposal showed that it was possible to use Pwntools to send gdb commands, which meant that I could simultaneously communicate with one instance of the vulnerable application with Pwntools and gdb from the same Python script. In practice however, this proved to be a lot harder than my initial research made it appear. Quite some time was spent on this part sadly.

In the end I got it working, but not in the most ideal way. Instead of switching between commands to the front and back-end, I decided to send one debug script when initially calling the application, that made sure the back-end data would get obtained. The rest of the interactions in my script are then for the front-end aspect of my exploit: actually sending the ROP chain payload and receiving the flag (or not). The register values after jumping to each gadget get saved to a text file, with the intention to use them as feedback for my reinforcement learning agent.

5.3 Which gadgets to provide to the RL agent?

Another aspect of automating our exploit is that we have to decide upfront which possible gadgets we want our reinforcement learning agent to consider when writing a possible exploit. As discussed in paragraph 2.1, the number of possible gadgets could get very big if we consider all of them. Since my simplified Q-learning approach would already increase the number of possible actions the agent could take (more on that in chapter 6), I decided for now to only provide a small subset of gadgets to the agent: three of the 'correct' gadgets and three more gadgets that should not be used in this exploit. In my final experiment, this would lead to one optimal action, 15 other successful actions and 1940 incorrect actions the RL agent could take.
I could have written a script that obtains these addresses automatically, but for the sake of development speed I used hardcoded addresses in my experiments.

6 Project: automating the reinforcement learning

Now that we can automatically attempt an exploit once, we now need to start many exploits, and learn from their results.
Here is where I suddenly realised a major flaw in my attempt at solving these challenges: in a 'normal' reinforcement learning process an agent only takes one step at a time, receives feedback on that step, and then decides on the next step based on that feedback. My plan was to see each gadget as one step. However with these ROP chain challenges there is only one opportunity to submit one or more gadgets: there is one vulnerable input that you can submit your exploit to, and that exploit either succeeds or fails. Therefore it is necessary to submit all gadgets the agent wants to try in one go.

I spent a lot of time searching for a reinforcement learning methodology that could do what I needed it to do: come up with a sequence of actions to try, try all of them in one go, analyse the results after each individual step, and then come up with a new sequence of actions based on the feedback from those previous actions. Things like actor-critic learning, offline/batch learning and model-based learning techniques sounded promising at first, but when I started to look deeper into their implementation it seemed like none of them provided what I was looking for. One of the difficulties of this part of the project stems from the apparent novelty of this approach. In particular, a thorough internet search and consulting others that are looking into reinforcement learning did not result in any implementable solutions that could come up with a sequence of actions upfront. It may be that little work is done in this area, or if there is it is not widely documented.

Therefore, in consultation with my supervisor, I decided to simplify my approach. Instead of considering each gadget as a separate action and needing the agent to come up with the correct sequence of actions, I've created those sequences myself by calculating all permutations of every possible size of the six gadgets I selected and then present these permutations to the agent as single separate actions. This means with 6 different gadgets I now have 1956 different actions: one single action that is the perfect correct action, 15 other actions that also solve the challenge as shown in Table 1, and

1940 actions that don't solve the challenge. Using this simplified approach means I lose a lot of relevant information about the challenge, but at least it did make for an executable setup.

Perfect action	**Actions that also lead to the flag**				
ABC	ABCD	ABCDE	ABCDF	ABCDEF	ABCEDF
	ABCE	ABCED	ABCEF	ABCDFE	ABCEFD
	ABCF	ABCFD	ABCFE	ABCFDE	ABCFED

Table 1: Possible solves - simplified

Code from Deeplizard (2018) was used to initialise this part of my code. My final code can be found on Github.

7 Project: experimenting and analysing

As discussed in chapter 3, for this project I experimented with changing the exploration decay rate as well as the reward. The other parameters stayed the same. Each experiment existed of 500 episodes, with a learning rate of 0.1, a discount rate of 0.99, an initial exploration rate of 1, and a max exploration rate of 1. This meant that for each experiment we would always start with exploring, since we don't have any knowledge to exploit yet. The varying exploration decay rate influenced how often the agent would go off exploring in later episodes.

While running the experiments I noticed that quite often an agent would consider one of the correct-but-not-optimal actions as the most optimal exploitation option, seemingly if the agent happened to run into one of these less optimal solutions earlier than the most optimal action, even if it did explore this optimal action later on in the process. I'll reflect on this in paragraphs 8.5 and 10.2.

For this report I'll analyse three 'good' setups and one 'less good' setup. This is by no means an exhaustive list of possible setups. The good setups were all run ten times, the other setup was run five times.

I ran my experiments with the settings as described in Table 2:

Setup	**Positive Reward**	**Exploration decay**	**Negative reward**	**Episodes**	**Experiments**
(a)	10	0.001	-1	500	5
(b)	10	0.005	-1	500	10
(c)	10	0.01	-1	500	10
(d)	4	0.01	-1	500	10

Table 2: Different experiment setups

Setup (a) didn't give very good results, which is why I only ran that experiment five different times.
A negative reward of -1 was given per gadget used, to encourage finding the solution

that takes the least number of steps. A reward of 10 was chosen to make a successful solve clearly stand out from unsuccessful solves. For setup (d) a positive reward of 4 was given for a successful solve, meaning that only the optimal solution (that requires three gadgets) would lead to a positive solution. All other solves (taking four, five or six gadgets) would lead to a reward of zero or lower in experiment (d).

8 Project: results

The code to create the charts in this chapter and the tables in Appendix B can be found on Github.

8.1 Setup (a)

For setup (a), I used a positive reward of +10 and an exploration decay rate of 0.001.

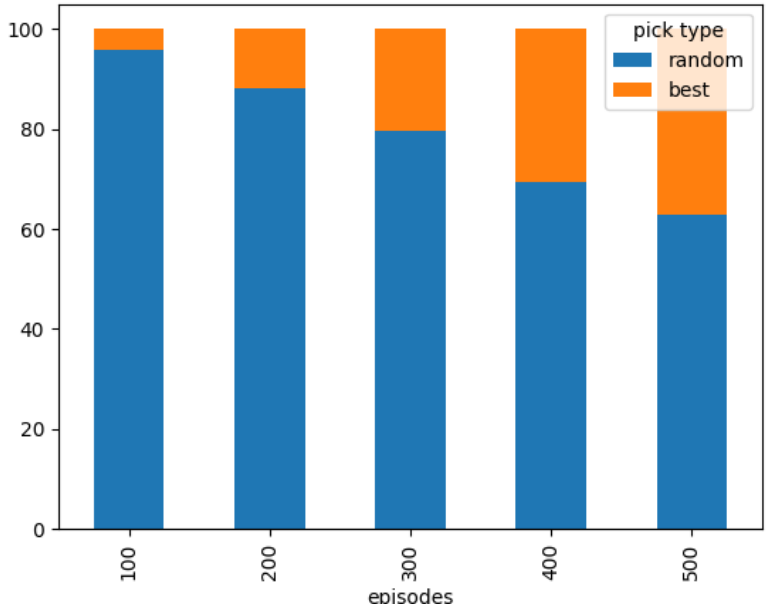


Figure 13: Setup (a): exploration vs exploitation per 100 episodes, in percentages

Figure 13 shows the percentage wise ratio of exploration vs exploitation per 100 episodes. This explains why experiment (a) didn't perform very well: even in episodes 401-500 the agent chose to perform a random action 62.8% of the time. This exploration decay rate clearly needed more episodes before it might converge into a 'useful' model. A combination of limited time available, the other experiment setups that did perform reasonably well after 500 episodes and the 'first results' after 500 experiments of setup (a) made me decide not to run experiment (a) with more than 500 episodes.

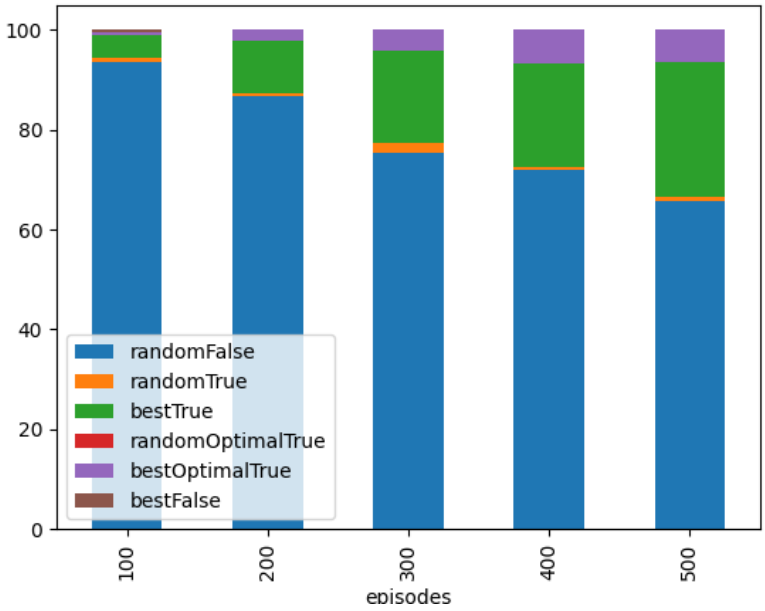


Figure 14: Setup (a): results per 100 episodes, in percentages

As shown in Figure 14, setup (a) tends to mostly end up preferring solutions that do solve the challenge, but aren't the optimal solution. This is something the other setups do as well, but obviously they converge faster since they have a faster exploration decay rate.

An explanation of the shorthand labels used in Figure 14, 16, 18 and 19:

- randomFalse stands for an exploration of an action that didn't lead to a solve;
- randomTrue stands for an exploration of an action that did lead to a solve;
- randomOptimalTrue stands for an exploration of the optimal solve;
- bestFalse stands for an exploitation of an action that didn't lead to a solve;
- bestTrue stands for an exploitation of an action that did lead to a solve;
- bestOptimalTrue stands for an exploitation of the optimal solve.

The exact numbers behind Figures 13-19 can be found in appendix B.

8.2 Setup (b)

For setup (b), I used a positive reward of +10 and an exploration decay rate of 0.005.

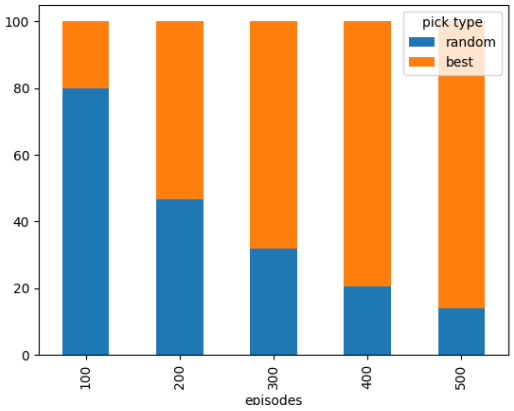


Figure 15: Setup (b): exploration vs exploitation per 100 episodes, in percentages

As shown in Figure 15, setup (b) the exploitation rate of 0.005 leads to almost 90% exploitation by episode 401-500, which is a big change over the 34% of exploitation towards the end of setup (a).

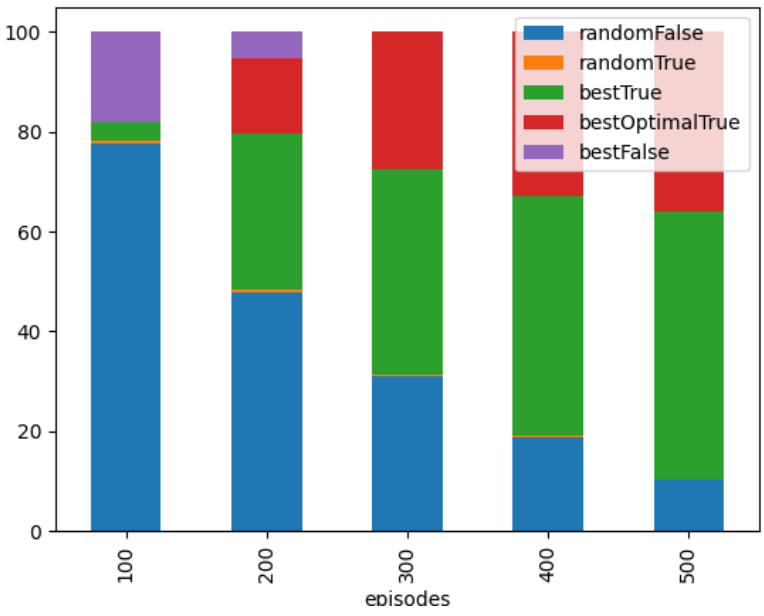


Figure 16: Setup (b): results per 100 episodes, in percentages

Figure 16 shows that during the final 100 episodes, 36% of all picks were exploiting the

best solve, and about 54% of all picks were exploiting another sub-optimal solve. In practice, this meant that 4 out of the 10 experiments converged to the optimal solve, and the other 6 to a sub-optimal solve.

8.3 Setup (c)

For setup (c), I used a positive reward of +10 and an exploration decay rate of 0.01.

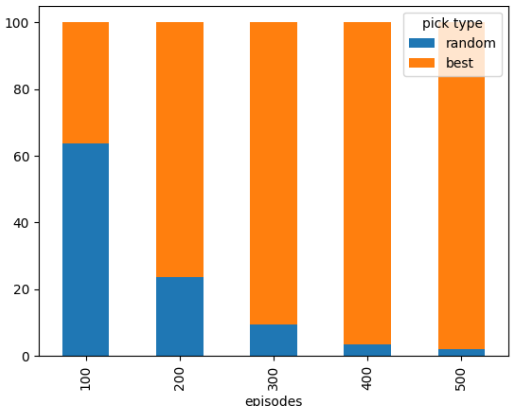


Figure 17: Setup (c): exploration vs exploitation per 100 episodes, in percentages

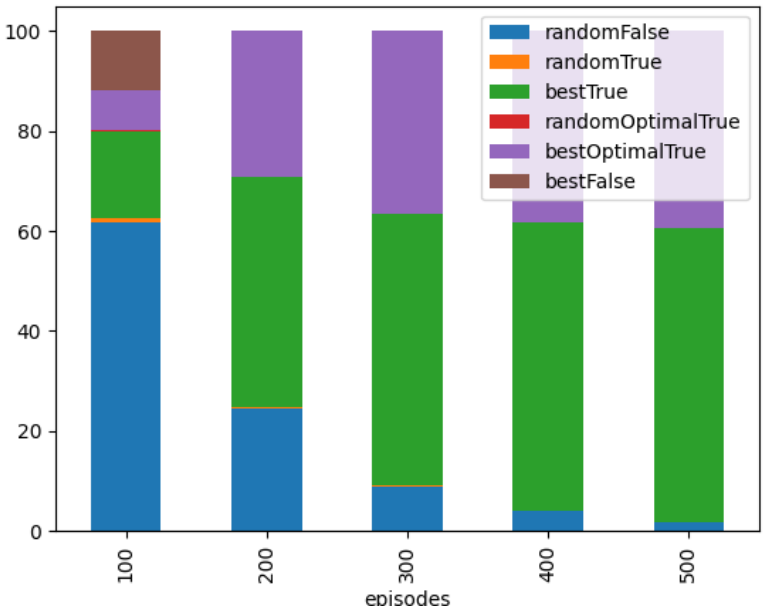


Figure 18: Setup (c): results per 100 episodes, in percentages

Setup (c) shows a very similar final 100 episodes, where about 40% of all picks were exploiting the best solve, and 59% were exploiting a sub-optimal solve. Here, again 4 experiments converged to the optimal solve and 6 to a sub-optimal solve. The difference in the percentages between setup (b) and (c) can be explained by the higher exploration decay rate in setup (c), which lead to more exploitation during those last 100 episodes.

8.4 Setup (d)

For setup (d), I used a positive reward of +4 and an exploration decay rate of 0.01. Since this is the same exploration decay rate as in setup (c), the percentages of exploration vs exploitation for setup (d) can also be seen in Figure 17.

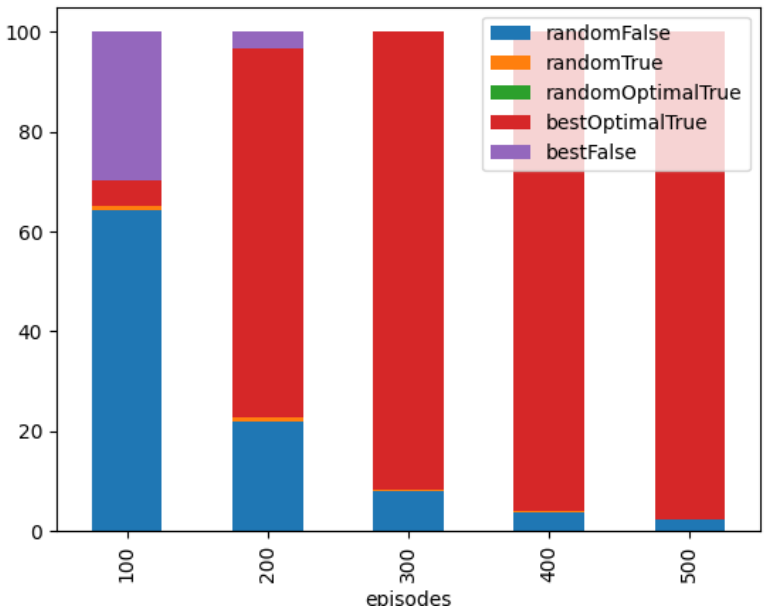


Figure 19: Setup (d): results per 100 episodes, in percentages

Here we can clearly see that choosing a positive reward in such a way that only the optimal solve leads to a positive final reward really helps the agent to converge to the optimal solve: after the first 200 episodes all 10 of the experiments only converge to the optimal solution.

8.5 Project: conclusions

My results show that experiment (d), with a positive reward of 4, leads to the most optimally successful agents. This does require the researcher to know the optimal length of the gadget chain in advance however, which is why I don't necessarily want to consider it the 'best' setup of this project.

In experiment (a) we can see that the optimal solution only gets chosen very sparsely, and we can see that it mostly favours other non-optimal solves as its 'best' exploitation option. Therefore I didn't see any advantages of the higher exploration rate that would make it worthwhile to continue experiment (a) for longer than 500 episodes to see if it would converge 'better' than the other three setups.

Experiment (b) and (c) also mainly choose a 'best' but not optimal solution when exploiting, but there is a chance the model converges to the 'correct' solution. Experiment (c) puts less focus on exploring than experiment (b), and therefore exploits its own learnings more after 500 episodes, resulting in a higher percentage of correct solutions. Experiments (b) and (c) got about the same percentage of conversion to the optimal solution (36 vs 39.3).

As stated before, experiment (d) requires to know the optimal number of gadgets needed to solve the challenge. Therefore this setup did the best for this specific challenge, but probably won't perform as good for other challenges that require a different length of ROP chain.

It can of course be questioned how essential it is to get to the most optimal solution. If the goal becomes to just find any solution that leads to obtaining the flag, we can say that setup (c), with a smaller exploration rate, leads to a higher number of episodes leading to a successful solution after 500 episodes, making setup (c) the best option of the four experiment setups explored for this project.

9 Conclusions

In my interim report I stated that to train my agent I was going to:

- Give all possible gadgets (lines of code to use) to the reinforcement learning environment (RLE).
- Have the RLE choose a gadget to use.
- Insert the output of the RLE into the whole code and run it against the CTF challenge.
- Pass the result back to the RLE. The result in this case will be whether the CTF was solved or not, and I'm currently planning on experimenting with returning other information like the value of (part of) the stack, the return pointer and/or other registers.
- Each gadget used in the ROP chain will give a small negative score (because the shorter the ROP chain the better) and solving the CTF obviously gives the highest score and ends the game. After a max number of times the game will also end. In versions where I also return other values, I might try to award small positive points for changing the return pointer to something other than the 'intended' address, although that might be very tricky to implement.

Here you can see that my thought process was slightly off when it gets to solving these challenges: you only get one moment to submit one or more challenges, which is not quite what I intended with the part of my fifth point where I stated "After a max number of times the game will also end". Once I combined all possible sequence into single actions, I mostly followed the steps as defined in my original methodology, I just didn't get to experiment with using the registers as further feedback for my agent.

When it gets to the end product, I intended to create a model that could test whether there is enough "useful existing code in the program that can be chained together" to achieve a goal such as finding a flag in a CTF or gaining root access in an application 'in the wild'.

In a way I achieved that, my agents do find a way to obtain the flag in the CTF challenge that's been given to them. However the resulting Q-table will only be applicable to the specific challenge the agent was trained on, it doesn't contain any knowledge that can be transferred to other challenges. Therefore I don't feel like I fulfilled that objective, but I've made the first steps towards fulfilling it.

10 Discussion

I'd like to think I made a good first start in automating ROP chain exploitation using reinforcement learning. Setting up the interaction between the actual ROP challenge and the reinforcement learning system proved to be quite challenging, but that base is there now. In this chapter I will reflect on some unexpected results and on the limitations of my current project. In the next chapter I will discuss how the next step, working with individual gadgets as individual actions, might be achieved.

10.1 Limitations of this project

The setup for this research was rather limited:

1. I used a very small subset of possible gadgets;
2. I was unable to use the results of each gadget that was used in the attack, instead I worked only with the final result (success/failure);
3. I only tried to find the solution for one challenge at a time, while you would want a more general model that can figure out how to solve several challenges.

Point number 2 in this case lead directly to point number 3. Therefore I believe point number 2 needs to be solved before further research into a more generalising model can make sense. As long as the agent only knows that option 37 is the correct answer it won't be able to come up with more generalising learnings: those should first come from learning what certain actions do to the environment, and then secondly from seeing that certain actions can have a similar result in a different environment.

10.2 Convergence

Besides realising I couldn't use the individual gadgets as my actions, I also noticed that quite often the agent would converge to a correct but sub-optimal solution. This was something I hadn't expected, but made me reflect on how I feel about that and how important converging to the optimal solution actually is.

C. J. C. H. Watkins (1989) states that optimal learning does not necessarily lead to the acquisition of the maximally efficient strategy: if learning the maximally efficient skill is costly, it may not be worthwhile to learn it. Translating that to my project, perhaps the goal shouldn't be to converge to the best option after several hundred episodes, but rather to quickly find at least one good option that solves the ROP chain challenge, since the best option can probably be distilled from it. In all of my setups, at least one 'best' option that leads to a solve is found after 200 episodes.
This could lead to a whole new approach to automating the actual use of the agent at least 'in the wild' (so after it has been trained), where as soon as a solve is found, a new process could kick-off that could iteratively remove the gadget at the end of the found sequence and then check if a solve is still achieved. Once a sequence no longer leads to a solve, the sequence before it might be considered the optimal solve.

Another thing that surprised me was that quite often the agent would exploit a good option as the 'best' option available, without first exploring that good option. I suspect this is because all options are instantiated at zero, and all bad options that get explored get a negative value, thus leaving all unexplored options at the 'best' score of zero and there being a chance of one of the unexplored good options to become chosen as the best option to exploit.

11 Suggestions for further research

11.1 Gadgets

In my current project I used specific memory addresses as the gadgets, but the same memory address will contain different gadgets (the actual code) in different applications and/or the same gadgets might be located at different addresses in different applications. Therefore it might be interesting in future research to not present the gadgets as memory addresses to the agent, but as the code behind the gadgets.

11.2 Hybrid learning

In order for my agent to use all information I have available after each gadget has been 'jumped to', I would need a reinforcement learning setup that can make an agent decide on a sequence of actions to take – and on how long that sequence of actions should be. For ease of writing, I'll call this form of reinforcement learning hybrid learning.
In this subsection I'll theorise on how such a system might work.

11.2.1 Other hybrid learning areas

As stated in paragraph 2.1, some ROP chain exploits require user input besides specific addresses in memory, such as typed text or integers. This means there are an infinite number of possible gadgets and therefore actions an agent might need to take to be successful, making ROP chain challenges not an ideal problem to solve using reinforcement learning.

There are however other situations, with a clearly defined limited set of possible actions, that would require hybrid learning to train an agent. Plenty of video games follow this principle for instance, such as card deck building games as Monster Train or Legends of Runeterra. In these games, the player needs to decide which of several cards to play, after which the round plays out and feedback on the performance of each individual card can be collected. The unique abilities of each card and the order in which these cards get played can be very important, as well as the optimal number of cards played.

Figure 20: Monster Train (Northernlion, 2020)

Several fighters can be positioned before a fight starts.

Small difference between these two games and the ROP chain attacks is that these games both have instant cards (throwing a fireball does instant damage to an enemy) as delayed cards (drafting a fighter won't do anything until the actual fighting round starts). For the sake of this chapter I will not make any further distinction between these types of cards, but consider playing those instant cards as separate rounds where the optimal sequence length should be one.

11.2.2 Deciding on the sequence length

Deciding on the optimal length of action sequence could be done with its own Q-learning process. The agent starts with a lot of experimenting with different sequence lengths, and gradually converges to an optimal length. For my current project, this would mean that since 0.83% of all sequences that are three, four, five or six gadgets long lead to a correct solve (as shown in Table 3), I would expect either of those four to become the favourite length of our agent depending on which length it finds its early successes with.

Chain length	Solves	Total	Percentage
3	1	120	0.83333
4	3	360	0.83333
5	6	720	0.83333
6	6	720	0.83333

Table 3: Successful solves as percentage of total number of permutations

One issue with this could be that different challenges require vastly different lengths, so it might be hard to create a base Q-table that could be used for multiple challenges. In the context of video games, the environment itself might give some relevant input that can help on deciding the optimal sequence length: in general it can be said that the further along in the game you get the harder the fights get, and several games show you at least some information on the enemies you will fight that round. Using this information when training your 'sequence length' agent might lead to decent results.

Another solution could of course be to not use reinforcement learning to determine the optimal sequence length, but instead determine the best option(s) through collecting and analysing the length of a lot of solves/wins in advance, if data like that is available.

One ultimate option in the case of ROP chain vulnerabilities is to only use the maximum possible length of a sequence as the length (so length six in my current project) since any 'extra' gadgets after a correct solve get ignored anyway. This loses all hope of finding the optimal solution of course, and might not work for other problems that would require sequence learning. For some games for instance this might lead to the agent wasting precious resources on earlier levels, making it impossible to win later rounds.

11.2.3 Deciding on the sequence

Once we've decided on the length of the sequence we want to try, we need to come up with a way to decide on the individual actions in the sequence. One way I envision this could be possible is to make a combination of 'online' learning (against the actual application) and offline learning:

1. We decide on the sequence length for this episode, let's call that number L;
2. We let an agent decide on an action to take (based on the usual exploration vs exploitation settings), but as long as that action isn't the Lth action, we run that action against our current 'offline' knowledge in the current Q-table;
3. We repeat the previous step until we've decided on the Lth action to take – then we combine all those actions together into one 'big' action we send to the actual environment;
4. We collect the feedback after each individual action, and update our policy accordingly;
5. If needed we update our 'deciding on sequence length' policy.

Figure 21 shows what this process might look like.

Step 2 could be either accomplished by creating a model of the environment and using that as the feedback towards our agent, or by looking up the current value for our state-action pair in a Q-table. Either way we might not get the correct result if we are exploring an action or a state-action pair we've never seen before, but 1) we don't have any better knowledge at that point and 2) we will update our 'offline' resource as soon as we've run the sequence against the environment, and then that knowledge can be used when deciding on the next sequence of events to try.

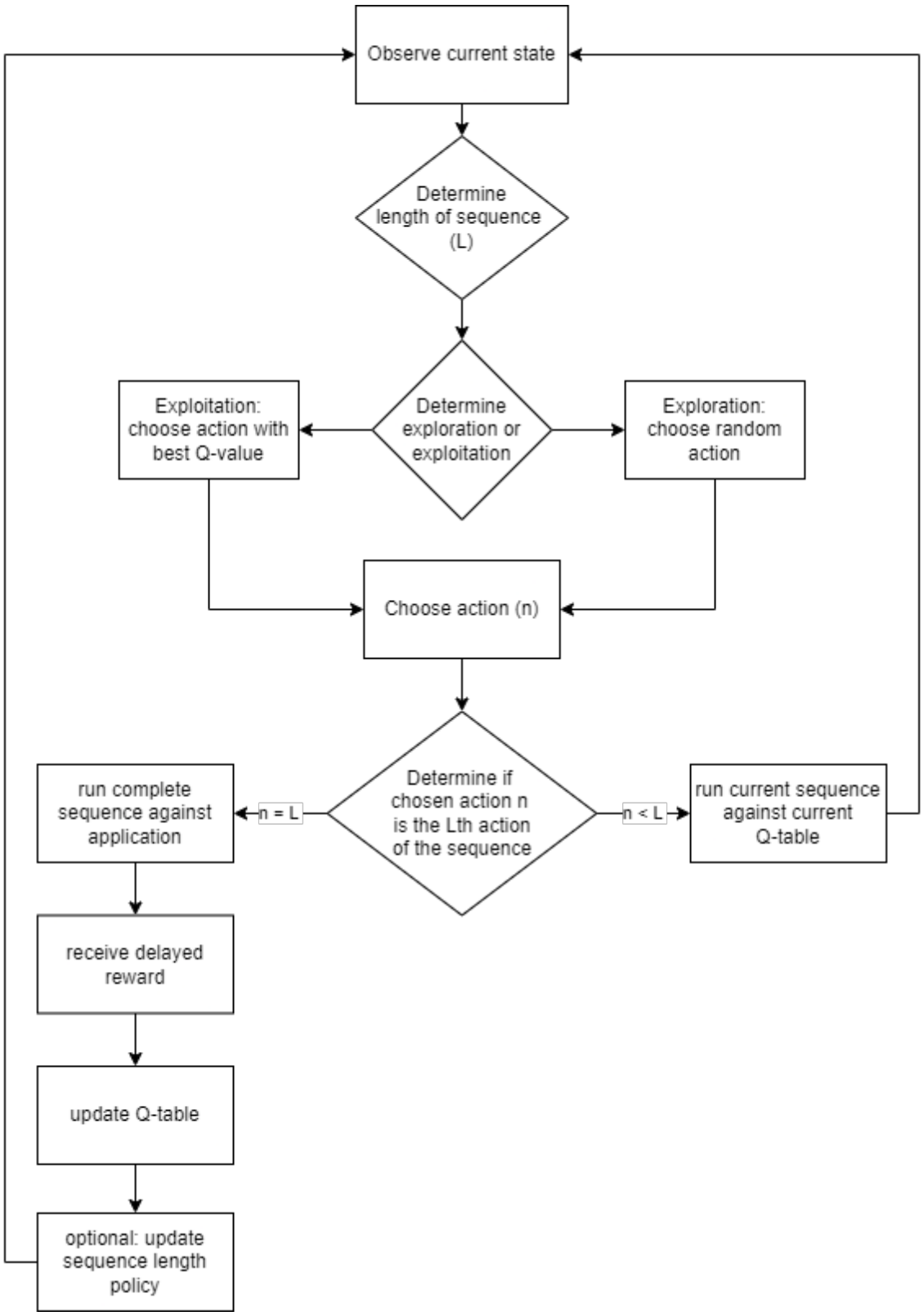


Figure 21: Hybrid learning process

12 References

Braga-Neto, U. (2020). *Fundamentals of pattern recognition and machine learning.* Springer.

Brunskill, E. (2019). *Stanford cs234: Reinforcement learning — winter 2019 — lecture 4 - model free control.* Retrieved October 20, 2021, from https://www.youtube.com/watch?v=j080VBVGkfQ

Brunton, S. L., & Kutz, J. N. (2021). *Data-driven science and engineering: Machine learning, dynamical systems, and control.* Cambridge University Press.

Chang, F., Chen, T., Su, W., & Alsafasfeh, Q. (2019). Charging control of an electric vehicle battery based on reinforcement learning. *2019 10th International Renewable Energy Congress (IREC)*, 1–63.

Clifton, J., & Laber, E. (2020). Q-learning: Theory and applications. *Annual Review of Statistics and Its Application*, *7*, 279–301.

Crutcher, P. D., Singh, N. K., & Tiegs, P. (2021). *Essential computer science: A programmer's guide to foundational concepts.* Apress L. P.

CTFtime team. (n.d.). *What is capture the flag?* Retrieved November 29, 2021, from https://ctftime.org/ctf-wtf/

Deeplizard. (2018). *Train q-learning agent with python.* Retrieved September 23, 2021, from https://deeplizard.com/learn/video/HGeI30uATws

Doeppner, T. (2019). *X64 cheat sheet.* Retrieved September 17, 2021, from https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

Gaskett, C., Wettergreen, D., & Zelinsky, A. (1999). Q-learning in continuous state and action spaces. *Australasian joint conference on artificial intelligence*, 417–428.

Gerg, I. (2005). An overview and example of the buffer-overflow exploit. *IAnewsletter, Spring.*

Goedegebure, C. (2020). *Buffer overflow attacks explained.* Retrieved September 18, 2021, from https://www.coengoedegebure.com/buffer-overflow-attacks-explained/

National Security Agency. (2021). *Ghidra.* Retrieved December 18, 2021, from https://github.com/NationalSecurityAgency/ghidra

Northernlion. (2020). *2020's best deckbuilder! — monster train (episode 1)*. Retrieved January 5, 2022, from https://www.youtube.com/watch?v=4H-Hez0zcOk

One, A. (1996). Smashing the stack for fun and profit. *Phrack magazine*, *7*(49), 14–16.

Roemer, R., Buchanan, E., Shacham, H., & Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, *15*(1), 1–34.

ROP Emporium. (n.d.). *Split*. Retrieved September 15, 2021, from https://ropemporium.com/challenge/split.html

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning*, *8*(3-4), 279–292.

Watkins, C. J. C. H. (1989). Learning from delayed rewards.

A Original planning

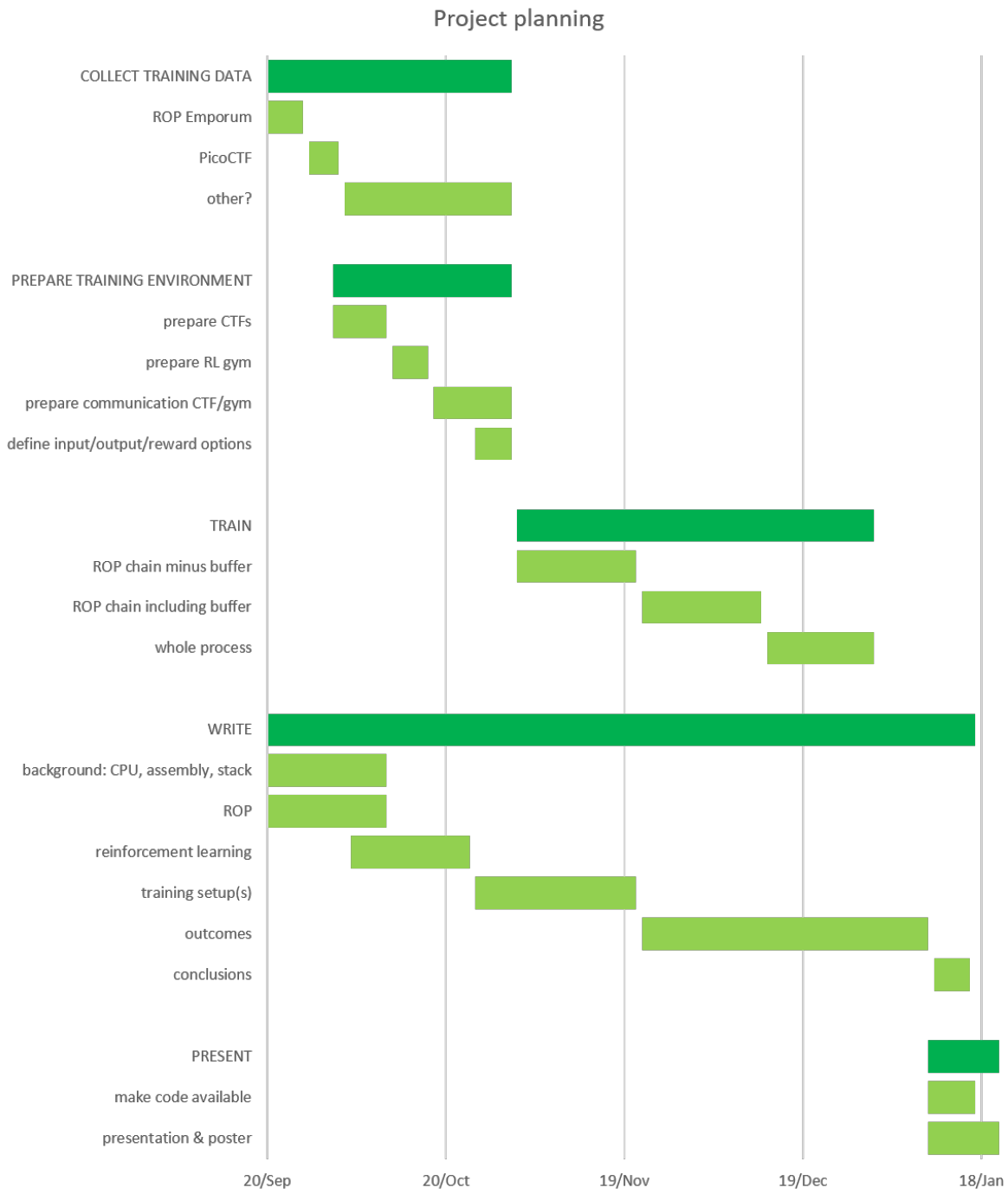


Figure 22: Original planning for the project

B Results

B.1 Exploration vs Exploitation

episodes	best	random
100	5.4	94.6
200	12.8	87.2
300	22.8	77.2
400	27.4	72.6
500	33.6	66.4

Figure 23: B.1 (a)

episodes	best	random
100	21.7	78.3
200	51.7	48.3
300	68.7	31.3
400	81	19
500	89.7	10.3

Figure 24: B.1 (b)

episodes	best	random
100	37.2	62.8
200	75.2	24.8
300	90.9	9.1
400	96	4
500	98.3	1.7

Figure 25: B.1 (c)

episodes	best	random
100	34.89	65.11
200	77.11	22.89
300	91.78	8.22
400	96	4
500	97.78	2.22

Figure 26: B.1 (d)

B.2 Pick percentages

randomFalse	randomTrue	bestTrue	randomOptimalT	bestOptimalTrue	bestFalse	episodes
93.6	0.8	4.4	0.2	0.6	0.4	100
86.6	0.6	10.6	0	2.2	0	200
75.4	1.8	18.6	0	4.2	0	300
72	0.6	20.6	0	6.8	0	400
65.6	0.8	27	0	6.6	0	500

Figure 27: B.2 (a)

randomFalse	randomTrue	bestTrue	bestOptimalTrue	bestFalse	episodes
77.7	0.6	3.6	0	18.1	100
47.7	0.6	31.2	15.1	5.4	200
30.9	0.4	41.1	27.6	0	300
18.8	0.2	48	33	0	400
10.2	0.1	53.7	36	0	500

Figure 28: B.2 (b)

randomFalse	randomTrue	bestTrue	randomOptimalT	bestOptimalTrue	bestFalse	episodes
61.6	1	17.4	0.2	7.8	12	100
24.5	0.3	46	0	29.2	0	200
8.9	0.2	54.2	0	36.7	0	300
3.9	0.1	57.8	0	38.2	0	400
1.7	0	59	0	39.3	0	500

Figure 29: B.2 (c)

randomFalse	randomTrue	randomOptimalTrue	bestOptimalTrue	bestFalse	episodes
64.22	0.78	0.11	5.22	29.67	100
22	0.89	0	73.89	3.22	200
8	0.22	0	91.78	0	300
3.89	0.11	0	96	0	400
2.22	0	0	97.78	0	500

Figure 30: B.2 (d)