CSci487 Penetration Testing Project: AILEE

Grant Haataja
UND Computer Science
Grand Forks, USA
grant.haataja@und.edu

David Wilson
UND Computer Science
Grand Forks, USA
david.andrew.wilson@und.edu

Michael Turnbull
UND Computer Science
Grand Forks, USA
michael.turnbull@und.edu

***Abstract*—This document details the planning, development, and workings of the penetration testing game AILEE, created as a final project for CSCI 487 Penetration Testing class at the University of North Dakota.**

I. Introduction

For this project on penetration testing topics, a hacking simulation game was created. The premise of the game is as follows: the user plays the role of a penetration-testing AI software named AILEE, which stands for Artificial Intelligence Linux Exploit Environment. The game takes place exclusively in a Linux-style terminal environment, with a limited arsenal of commands for the player to use. As the player progresses through the game and "learns" as an AI, the commands available for use increase. Throughout the game, the player is given typed instructions and information from the AI's administrator to assist in learning.

There are two targets to hack in this demo, although there is much potential for expansion. The game uses simulated port scanning, vulnerability scanning, exploitation, and other penetration testing tools to mimic real-life penetration testing methods. Additionally, the game features a storyline with three possible endings, depending on player actions. Special care was taken to handle proper sequence of events.

II. Investigation

A. Planning the Project

Before beginning the development of the game, a suitable platform to run the environment needed to be found. The website Repl.it was decided upon, due to their extensive language support and the ability for multiple people to work simultaneously and have all changes automatically saved to the cloud. [1] The "Multiplayer" mode, as this feature was called, still had a lot of bugs, so forking the project and saving work manually was still necessary, but overall it made the development of AILEE much smoother.

Python3 was selected as the programming language of choice, due to its ease of scripting and strong object-oriented nature. The various classes corresponding to different aspects of the game and environment would be programmed separately, as well as Python scripts for each command available to the player, and every storyline event that could be run. The original plan was for there to be three different targets for the player to hack, but due to time limitations the scope was decreased to two targets.

To enable smooth graphics for the intro screen and the game's ending events, the Python Curses library was referenced and used extensively. [2] This provided the ability to control keyboard input while text displayed on the screen or the ending event graphics played, to increase the smoothness of gameplay.

III. Project Description

A. Intro Screen

For the graphics of the intro screen for the game, ASCII art was used to spell the word AILEE, along with a selection for New Game or Exit. The user can move between the selection using the up or down arrow keys and choose by pressing the enter key. Selecting Exit will cause the terminal session within Repl.it to exit and the game will have to be run again, selecting New Game creates a new session and runs the game.

In addition to these, pressing the up arrow six times in a row will show a hidden third selection, Skip Dialog. This will run the game without displaying any of the instructions and information from the administrator to AILEE, and was very useful for testing the game during development. This mode is not explained or mentioned in the game, as it is not recommended to play without reading the dialogue.

The intro screen makes use of the Python curses library to allow smooth use of the arrow keys keyboard input and prevent buggy graphics.

B. Starting the Game

Upon choosing New Game, the user watches as the administrator logs into their account and launches AILEE.exe to start a new shell. After the shell loads, the first event triggers and text displays on the screen to inform the user what is going on. The administrator gives a brief explanation, and then the user is free to experiment with the Linux-style terminal environment. The terminal runs in the Shell class, (in tandem with the Game and DoStory classes), which supports multiple terminals on various computers. The code for the Shell class is as follows:

```
from termcolor import colored
import replit

import time
import traceback, sys, random

import executables
import events
from MainMenuException import MainMenuException
```

```

DEFAULT_PROMPT = colored("AILEE@{COMP}: {CWD}$ ", 'green')

CMD_NOT_FOUND_STRS = [
  "command not found"
]

class Shell(object):
  """
  Like a seashell.
  """

  def __init__(self, computer, user, agent=None, cwd=None, game=None):
    """
    Create a shell.
    """

    self.computer = computer
    self.user = user
    self.agent = agent
    self.cwd = cwd or computer.fs
    self.prompt = DEFAULT_PROMPT
    self.running = False
    self._command_dictionary = {}
    self.variables = {}
    self.game = game
    self.history = []

    self._setup()

  def _setup(self):
    replit.clear()
    s = "Loading new shell"

    print(s, end='\r')
    i = 0

    # load command dictionary
    for module in executables.__all__:
      self._command_dictionary.update({module: getattr(executables, module).run})
      print(s + '.'*i, end='\r')
      i += 1
      time.sleep(0.1)
    time.sleep(0.3)
    replit.clear()
    #print(constants.title)

  def _get_command_from_str(self, command_str):
    """
    Takes a command name, returns the executable object.
    """

    if command_str == '':
      return False
    if command_str not in self.game.allowed_commands:
      return None
    cmd = self._command_dictionary[command_str]
    return cmd

  def run_command(self, command, args):
    """
    Runs a command.

    Input must be a runnable command that accepts **kwargs.
    """

    command(
      *args,
      computer=self.computer,
      cwd=self.cwd,
      user=self.user,
      agent=self.agent,
      shell=self,
      game=self.game,
    )

  def take_input(self):
    user_input = input(self.prompt.format(
      COMP=str(self.computer.name),
      CWD=str(self.cwd),
      USER=self.user),
    )

    parts = [p.strip() for p in user_input.split(' ')]
    command = parts[0]
    args = parts[1:]

    return command, args

  def one_command(self):
    command, args = self.take_input()
    cmd = self._get_command_from_str(command)
    if cmd is None:
      self.cmd_not_found()
      return
    elif cmd is False:
      return  # nothing on empty commands
    cname = cmd.__module__.split('.')[-1]
    if not cname == 'doStory':
      self.game.history.append([cname, args])
      self.history.append([cname, args])

    if not (command or args):
      return  # skip empty input
    self.run_command(cmd, args)

  def halt(self):
    self.running = False

  def cmd_not_found(self):
    self.history.append([None, []])
    self.game.history.append([None, []])
    print("Command not found")
    #print(random.choice(CMD_NOT_FOUND_STRS))

  def start_shell_loop(self):
    self.running = True
    while self.running:

      # This is the line of code that integrates the story VVV


      try:
        self.run_command(events.doStory.run, [])
        self.one_command()
      except KeyboardInterrupt:
        print()
        #print("\nYou can't leave! ", end='')
      except KeyError as e:
        self.cmd_not_found()
      except AssertionError as e:
        print(str(e))
      except MainMenuException:
        raise MainMenuException
      except Exception as e:
        print(colored("Something went wrong.  I'm not quite sure what.  Maybe try again?", 'red'))
        # Uncomment VV for full tracebacks
        #einfo = sys.exc_info()
        #traceback.print_exception(*einfo)
```

The user is encouraged to try out the various possible commands, which can be displayed using the *help* command. The story continues after the user has ran ten commands, (they can be the same or different commands, it doesn't matter).

C. Gameplay and Executables

As the game progresses, the user will utilize the available penetration testing tools to gain access to the target computers. There are tools for finding IP addresses, port scanning, vulnerability scanning, exploitation, password cracking, and connecting through the ftp file sharing network. Generally, some of the commands require the results from running other commands in order to be run successfully.

In particular, the *exploit* command is extremely useful for gaining access to other computers. This executable is similar to the Metasploit framework which is commonly used in real-life penetration testing. In the game, it only has two possible exploits to choose from, and if ran against a vulnerable IP address with the proper port selection, will successfully open a new shell on the target computer.

```
'''
"Start the exploitation station."

Description: exploit is a software used for gaining
    unauthorized access to remote\ncomputers. There
    are different exploits within the framework for
    the user to choose\nfrom. Upon running the
    exploit software, user will need to choose the
    exploit to\ntry, enter the target IP address,
    enter the port to connect to, and then type\n"
    run" to attempt the exploit.

Usage: exploit
'''
#Would like 2 different exploit options to start
    with
#one for 'windoors' systems and one for 'lionux'
    systems

from funfunctions import dots

def run(*args, **kwargs):
  emptyList = True
  for arg in args:
    if arg:
      emptyList = False
  assert len(args) == 0 or emptyList, "Invalid use
    of exploit.\n\nUsage: exploit"

  print('***Welcome to the exploitation station***')
  print('Available exploits:')

  exploits = {
    1: 'WD45_702 reverse tcp shell',
    2: 'LI38_612 meta ssh security flaw',
  }

  for i, exploit in exploits.items():
    print("{:2d}. {}".format(i, exploit))

  sel = 0
  while not sel in exploits.keys():
    try:
      sel = int(input("Exploit selection > "))
    except ValueError:
      print("Enter a number")

  addr = input("Enter target IP address > ")
  port = -1
  while port < 0:
    try:
      port = int(input("Select port to use > "))
    except ValueError:
      print("Enter a number")
  run = input("Type 'run' to begin expoitation > ")

  if run != 'run':
    return

  dots("Running exploit", 9, 0.333)

  if sel == 1:
    if (addr == '120.45.30.6') and (port == 1100)
    and (run == 'run') and kwargs['game'].network['
    120.45.30.6']:
      print("Exploit success.")
      kwargs['game'].network['120.45.30.6'].
    exploited = True
      kwargs['shell'].run_command(
        kwargs['shell']._get_command_from_str('shell
    '),
        ['new', '120.45.30.6']
      )
    else:
      # the failure mode
      print("Exploit failed.")


  elif sel == 2:
    # run exploit 2
    if (addr == '120.33.7.242') and (port == 22) and
     (run == 'run'):
      print("Exploit success.")
      kwargs['game'].network['120.33.7.242'].
    exploited = True
      kwargs['shell'].run_command(
        kwargs['shell']._get_command_from_str('shell
    '),
        ['new', '120.33.7.242']
      )
    else:
      # the failure mode
      print("Exploit failed.")
```

Above is the code for the *exploit* executable in the game. In terms of options, it does not compare to the Metasploit framework, but the goal was to create it to feel similarly in the terminal environment.

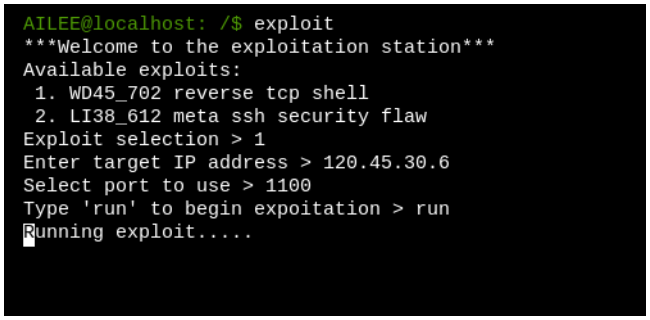

Fig. 1. Running the *exploit* executable

First, the user enters the number corresponding to the exploit they wish to run. Then, the target IP address is entered, followed by the specific port, and finally the command 'run'

must be entered and the exploitation software will attempt to gain access to the target (Fig. 1).

Fig. 2. Gaining a shell on target machine

If the exploit is successful, the screen will clear and the words ”Loading new shell....” will appear on the screen with increasing dots as the shell loads (Fig 2).

Fig. 3. A new shell on an exploited target

Once the shell loads, the name of the computer exploited will be shown before the /$ symbol where commands are typed (Fig 3).

Most of the other executables play an important role in the game, with a few exceptions that were included for comedic value.

D. Events and Storyline

The story development of the game is controlled by event scripts that are triggered at specific times. Running the events in the proper order is critical for the game to play as planned. The first event is run immediately after the game loads its first shell on the localhost computer. Each event after that has a condition that must be met to trigger it. The second event runs after ten commands have been run, the third event runs after the *pscan* command runs, and so on. Each event runs instructions for the user that should allow them to trigger the next event and progress in the game.

```
#Third dialogue of the game
#triggers after port scanning has been done
import time
from funfunctions import typewriter
from termcolor import colored

def check_run(*args, **kwargs):
  # check for 'pscan' in history, most recently run, and exactly once
  if len(kwargs['game'].history) == 0:
    return False
  if not 'event2' in kwargs['game'].events_run:
    return False

  command = ['pscan', ['120.45.30.6']]

  a = kwargs['game'].history[-1] == command
  return a

def run(*args, **kwargs):
  # using kwargs we can get access to the shell, and from within the event
  # have the user run commands

  color = 'cyan'
  game = kwargs['game']

  text = [
    '\nGood job, Ailee. I see you have successfully found which open ports are running\non our target.\n\n',
    'Our next step is to run our vulnerability scanning software against the target\nto see if we can use any exploits against them.\n\n'
  ]
  filename = 'message03.txt'
  if filename not in game.eventLogDir:
    game.eventLogDir.addFile(filename, colored(''.join(text), color))

  if not game.skip_dialog:
    typewriter(colored(text[0],color))
    typewriter(colored(text[1],color))
  else:
    print(colored('Event3 text skipped', 'red'))

  # Create chat log in AILEE's directory
```

This is how the scripts for the events are written, implementing a custom-built ”typewriter” function created to display text to the screen word by word as if it were typed, and using a blue color to differentiate it from the rest of the game text.

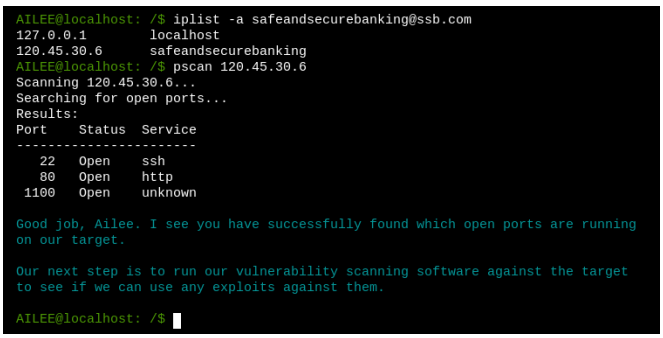

Fig. 4. The dialogue for event 3

E. Some Common Mistakes

- The word “data” is plural, not singular.
- The subscript for the permeability of vacuum μ_0, and other common scientific constants, is zero with subscript formatting, not a lowercase letter “o”.

- In American English, commas, semicolons, periods, question and exclamation marks are located within quotation marks only when a complete thought or name is cited, such as a title or full quotation. When quotation marks are used, instead of a bold or italic typeface, to highlight a word or phrase, punctuation should appear outside of the quotation marks. A parenthetical phrase or statement at the end of a sentence is punctuated outside of the closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses.)
- A graph within a graph is an "inset", not an "insert". The word alternatively is preferred to the word "alternately" (unless you really mean something that alternates).
- Do not use the word "essentially" to mean "approximately" or "effectively".
- In your paper title, if the words "that uses" can accurately replace the word "using", capitalize the "u"; if not, keep using lower-cased.
- Be aware of the different meanings of the homophones "affect" and "effect", "complement" and "compliment", "discreet" and "discrete", "principal" and "principle".
- Do not confuse "imply" and "infer".
- The prefix "non" is not a word; it should be joined to the word it modifies, usually without a hyphen.
- There is no period after the "et" in the Latin abbreviation "et al.".
- The abbreviation "i.e." means "that is", and the abbreviation "e.g." means "for example".

An excellent style manual for science writers is [**?**].

F. Authors and Affiliations

The class file is designed for, but not limited to, six authors. A minimum of one author is required for all conference articles. Author names should be listed starting from left to right and then moving down to the next line. This is the author sequence that will be used in future citations and by indexing services. Names should not be listed in columns nor group by affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization).

G. Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is "Heading 5". Use "figure caption" for your Figure captions, and "table head" for your table title. Run-in heads, such as "Abstract", will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced.

H. Figures and Tables

a) Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation "Fig. 4", even at the beginning of a sentence.

TABLE I
TABLE TYPE STYLES

Table Head	**Table Column Head**		
	Table column subhead	***Subhead***	***Subhead***
copy	More table copy[a]		

[a]Sample of a Table footnote.

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity "Magnetization", or "Magnetization, M", not just "M". If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write "Magnetization (A/m)" or "Magnetization {A[m(1)]}", not just "A/m". Do not label axes with a ratio of quantities and units. For example, write "Temperature (K)", not "Temperature/K".

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

Please number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [**?**]—do not use "Ref. [**?**]" or "reference [**?**]" except at the beginning of a sentence: "Reference [**?**] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [**?**]. Papers that have been accepted for publication should be cited as "in press" [**?**]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [**?**].

REFERENCES

[1] Repl.it, "The world's leading online coding platform," repl.it. [Online]. Available: https://repl.it/site/features. [Accessed: 02-May-2019].

[2] A. M. Kuchling and E. S. Raymond, "Curses Programming with Python¶," Curses Programming with Python - Python 3.7.3 documentation. [Online]. Available: https://docs.python.org/3/howto/curses.html. [Accessed: 02-May-2019].

IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper prior to submission to the conference. Failure to remove the template text from your paper may result in your paper not being published.