CSci487 Penetration Testing Project: AILEE

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

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

Michael Turnbull
UND Computer Science
Monroe, NH, 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 AIs 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 games 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:

```python
# -*- coding: utf-8 -*-

from termcolor import colored
import funfunctions

import time
import traceback
import sys
import random

import executables
import events
from MainMenuException import MainMenuException

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

CMD_NOT_FOUND_STRS = [
    "command not found",
    "Nope, don't know that one",
    "This isn't Google",
    "NOOB!"
    "Segmentation fault (core dumped)",

]


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):
        funfunctions.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)
        funfunctions.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, **kwargs):
        """
        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,
            **kwargs
        )

    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:
            try:
                self.run_command(events.doStory.run, [])
                self.one_command()
            except KeyboardInterrupt:
                print()
            #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).

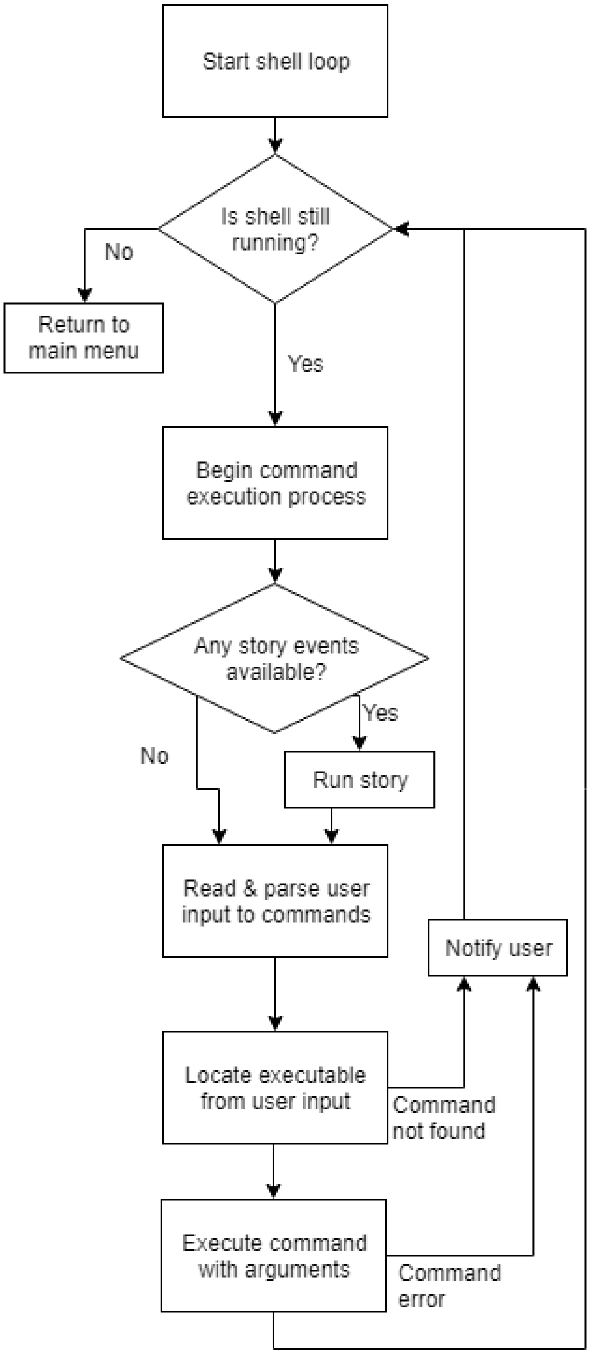

Start shell loop
Is shell still running?
No
Return to main menu
Yes
Begin command execution process
Any story events available?
No
Yes
Run story
Read & parse user input to commands
Notify user
Locate executable from user input
Command not found
Execute command with arguments
Command error

Fig. 1. Shell flow diagram

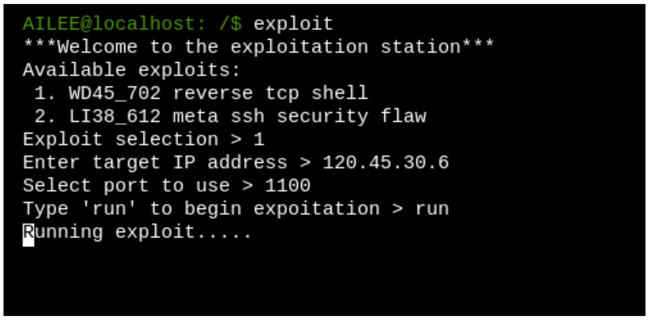
C. Gameplay and Commands

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 command is similar to the Metasploit framework, commonly used in real-life penetration testing. In the game, as the player uncovers exploits in different computers with the `vscan` command, those exploits are added to a global database of exploits which the user can then pick from in the `exploit` command. Exploits are specific to a port on a computer, and if correctly used, 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
computers. There are different exploits within the framework for the user to choose
from. Upon running the exploit software, user will need to choose the exploit to
try, enter the target IP address, enter the port to connect to, and then type
”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:’)

    vdb = kwargs[’game’].vuln_database
    visible_exploits = [name for name in vdb if not name.startswith(’_’)]
    exploits = {i+1: vdb[i] for i in range(len(visible_exploits))}

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

    sel = 0
    while sel not 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")
    chkrun = input("Type 'run' to begin exploitation > ")

    if chkrun != 'run':
        return

    dots("Running exploit", 9, 0.333)

    try:
        box = kwargs['game'].network[addr]
        vuln = exploits[sel]
    except KeyError:
        print("Invalid options specified.")
        return

    if (vuln in box.vulns) and \
       (port == box.vulns[vuln][1]):
        box.vulns[vuln][0] = True
        print("Exploit success!")
        kwargs['shell'].run_command(
            kwargs['shell']._get_command_from_str('shell'),
            ['new', addr]
        )
    else:
        print("Exploit failed")
```

Above is the code for the *exploit* command 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.



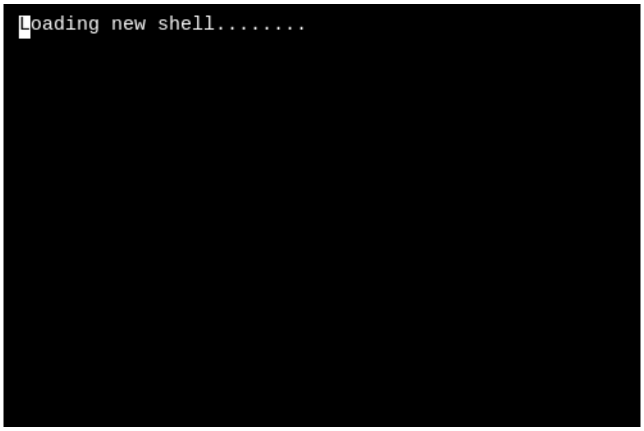
```
AILEE@localhost: /$ exploit
***Welcome to the exploitation station***
Available exploits:
 1. WD45_702 reverse tcp shell
 2. LI38_612 meta ssh security flaw
Exploit selection > 1
Enter target IP address > 120.45.30.6
Select port to use > 1100
Type 'run' to begin expoitation > run
Running exploit.....
```

Fig. 2. 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. 2)

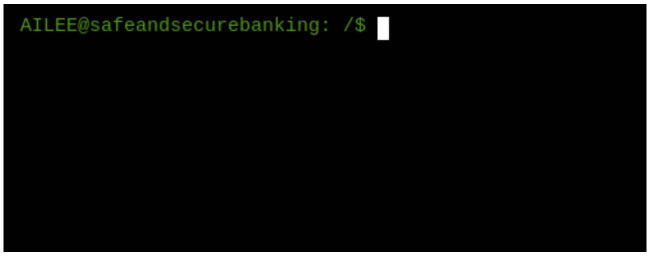
If the exploit is successful, the screen will clear and the words "Loading new shell...." will appear on the screen with increasing dots as the shell loads (Fig. 3)

Once the shell loads, the name of the computer exploited will be shown before the /$ symbol where commands are typed (Fig. 4).



```
Loading new shell........
```

Fig. 3. Gaining a shell on target machine



```
AILEE@safeandsecurebanking: /$
```

Fig. 4. A new shell on an exploited target

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

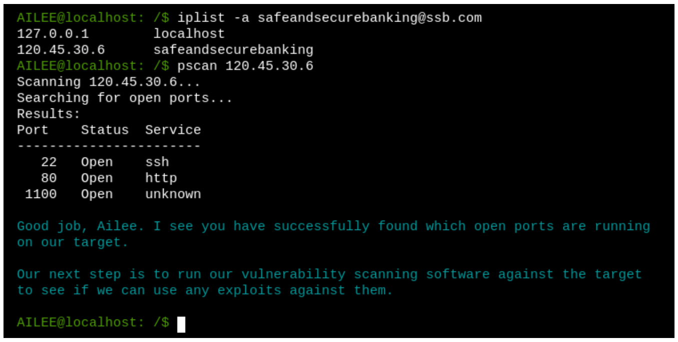
D. Events and Storyline

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.



```
AILEE@localhost: /$ iplist -a safeandsecurebanking@ssb.com
127.0.0.1        localhost
120.45.30.6      safeandsecurebanking
AILEE@localhost: /$ pscan 120.45.30.6
Scanning 120.45.30.6...
Searching for open ports...
Results:
Port    Status  Service
-----------------------
  22    Open    ssh
  80    Open    http
 1100   Open    unknown

Good job, Ailee. I see you have successfully found which open ports are running
on our target.

Our next step is to run our vulnerability scanning software against the target
to see if we can use any exploits against them.

AILEE@localhost: /$
```

Fig. 5. The dialogue for event 3

This is the text for the third event (Fig. 5), which triggers after the user runs the command *pscan* against the first target IP address.

There are three special events that do not trigger in a normal play-through of the game. These events will only trigger if the player thinks for themselves and uses information found in the game creatively and without instruction from the administrator. Interestingly, if the user follows the instructions without diverging on their own, they will lose the game during the final hack. The only way to win is by finding what the information leads to and changing the fate of the game before attempting to exploit the final target.

E. Computers and Filesystems

AILEE uses a filesystem structure that feels like a Linux terminal. There are a total of four computers in the game, localhost (AILEE's home computer), the two targets, and one other computer. Each computer has a unique filesystem, with directories and files. As time was limited, the *bin* and *log* directories were left empty on all computers, but the *home* directories all have files in them that either pertain to the gameplay or exist for comedic value or storyline development.

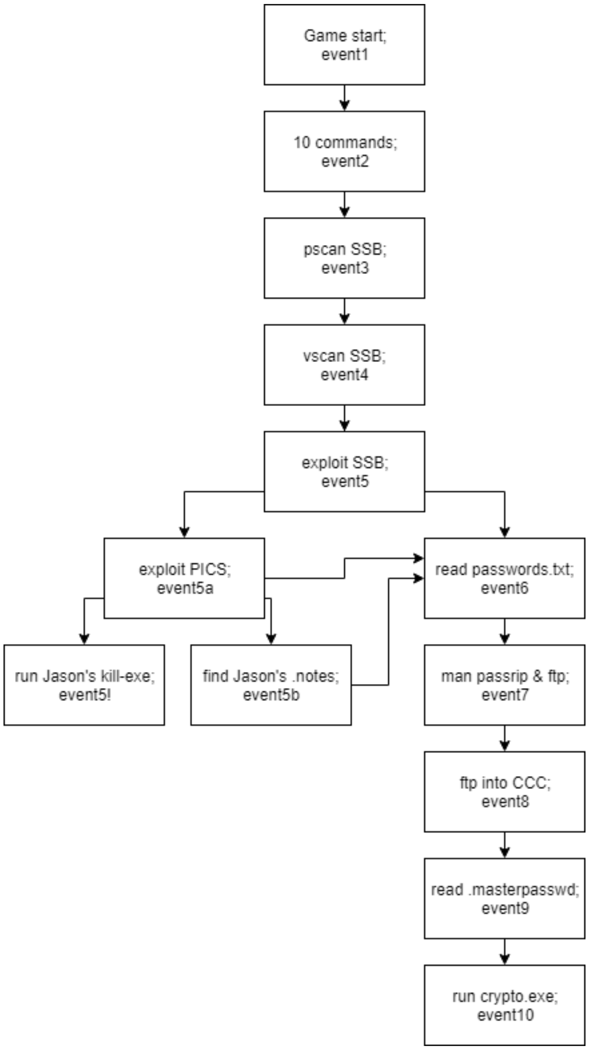


Fig. 6. The sequence of events in play order

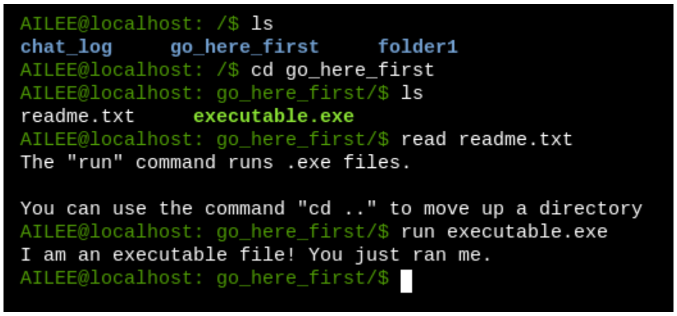
To navigate the filesystems, the traditional Linux commands *ls* and *cd* are used, as well as the commands *read* and *run*, which display file text and run executable files, respectively.

```
import hashlib


class Permissions(object):
    """
    rwxrwx
     |||||\ Owner exec
     ||||\ Owner write
     |||\ Owner read
     ||\ General user exec
     |\ General user write
     \ General user read
    """

```

```python
    def __init__(self, perms):
        self._bits = perms
        assert len(self._bits) == 6

    def _set_bit(self, n, newval):
        bits = list(self._bits)
        bits[n] = newval
        self._bits = ''.join(bits)

    def __str__(self):
        return self._bits

    def __eq__(self, other):
        if isinstance(other, Permissions):
            return self._bits == other._bits
        elif isinstance(other, str):
            return self._bits == other
        else:
            return False

    def __hash__(self):
        return hash(self._bits)

    def __getitem__(self, bit):
        return self._bits[bit]

    def __setitem__(self, bit, newval):
        self._set_bit(bit, newval)

    @property
    def read_users(self):
        return self._bits[0] == 'r'

    @read_users.setter
    def read_users(self, allowed):
        self._set_bit(0, 'r' if allowed else '-')

    @property
    def write_users(self):
        return self._bits[1] == 'w'

    @write_users.setter
    def write_users(self, allowed):
        self._set_bit(1, 'w' if allowed else '-')

    @property
    def exec_users(self):
        return self._bits[2] == 'x'

    @exec_users.setter
    def exec_users(self, allowed):
        self._set_bit(2, 'x' if allowed else '-')

    @property
    def read_owner(self):
        return self._bits[3] == 'r'

    @read_owner.setter
    def read_owner(self, allowed):
        self._set_bit(3, 'r' if allowed else '-')

    @property
    def write_owner(self):
        return self._bits[4] == 'w'

    @write_owner.setter
    def write_owner(self, allowed):
        self._set_bit(4, 'w' if allowed else '-')

    @property
    def exec_owner(self):
        return self._bits[5] == 'x'

    @exec_owner.setter
    def exec_owner(self, allowed):
        self._set_bit(5, 'x' if allowed else '-')


class Directory(object):
    """
      Tree structure of directories and files
    """
    def __init__(self, name=None, parent=None, children=None, permissions='r-xr-x',
                 owner=None):
        self.name = name or ''
        if parent is None:
            self.parent = self
        else:
            self.parent = parent
        self.permissions = Permissions(permissions)
        self.owner = owner or 'n/a'
        self.children = {
            '.': self,
            '..': self.parent
        }
        if type(children) is dict:
            self.children.update(children)

    def mkdir(self, name, **kwargs):
        assert (name not in self.children), 'Directory already exists'
        newDir = Directory(name=name, parent=self, **kwargs)
        self.children.update({name: newDir})
        return newDir

    def addFile(self, fileName, fileContents, **kwargs):
        assert (fileName not in self.children), 'File already exists'
        assert ('.' in fileName), 'File has no type'
        newFile = File(fileName, fileContents, **kwargs)
        self.children.update({fileName: newFile})
        return newFile

    def addPrebuiltFile(self, file, **kwargs):
        assert (file.name not in self.children), "File already exists"
        self.children.update({file.name: file})
        return file

    def rmFile(self, fileName):
        assert (fileName in self.children), "File does not exist"
        assert ('.' in fileName), "File has no type"
        del self.children[fileName]

    def __iter__(self):
        return (fName for fName in self.children)

    def __repr__(self, base=True):
        output = ""
        if self.parent is not self:
            up1 = self.parent
            while isinstance(up1, Directory):
                output = up1.name + "/" + output
                if up1 is up1.parent:
                    break
                else:
                    up1 = up1.parent
        output += self.name
        if base:
            output += '/'
        return output

    __str__ = __repr__  # set __str__ as the same
    method as __repr__

    def __getitem__(self, item):
        assert (item in self.children), "File or Directory not found"
        return self.children[item]

    def __len__(self):
        # Remove the "." and ".." folders from the size
        # They're not really there
        return len(self.children) - 2


class File(object):
    '''
      Stores things. Like data, machine code, and blackmail
    '''
    def __init__(self, name, data='', permissions='r--r--', owner=None, **kwargs):
        self.name = name
        self._data = data
        self.permissions = Permissions(permissions)
        self.owner = owner or 'n/a'
        self._original_hash = hashlib.md5(data.encode('utf-8')).hexdigest()
        self._current_hash = hashlib.md5(data.encode('utf-8')).hexdigest()
        self._kwargs = kwargs

    def append(self, data):
        self._data += data
        self._current_hash = hashlib.md5(self._data.encode('utf-8')).hexdigest()

    @property
    def original_hash(self):
        return self._original_hash

    @property
    def current_hash(self):
        return self._current_hash

    @property
    def data(self):
        return self._data

    @data.setter
    def data(self, newdata):
        self._data = newdata
        self._current_hash = hashlib.md5(self._data.encode('utf-8')).hexdigest()

    def __repr__(self):
        return self.name
    __str__ = __repr__ # set __str__ as the same method as __repr__

    def __len__(self):
        return len(self._data)
```

This is the code for the structure and mechanics of the computer filesystems. Similar to Linux, both files and folders have and use discrete permissions, marking certain files as readable, writable, and executable. While there are not yet any provisions for editing/creating files in-game[1], the permissions architecture is in place for adding future file-editing utilities.

Executable files are created from a separate filesystem constructor. Each executable file simply is a `File` object with the executable bit set in its permissions object, and the file contents are pure Python code that executes when the file is run via the `run` command. To verify that arbitrary code is not run (which would break the game), executable files compare hashes from creation to runtime to and will not run if the contents have been altered.



```
AILEE@localhost: /$ ls
chat_log      go_here_first      folder1
AILEE@localhost: /$ cd go_here_first
AILEE@localhost: go_here_first/$ ls
readme.txt      executable.exe
AILEE@localhost: go_here_first/$ read readme.txt
The "run" command runs .exe files.

You can use the command "cd .." to move up a directory
AILEE@localhost: go_here_first/$ run executable.exe
I am an executable file! You just ran me.
AILEE@localhost: go_here_first/$
```

Fig. 7. Filesystem navigation

Directories are colored light blue, executables are light green, and regular files are white. Because of the permissions associated with each file and directory, and the fact that `File` and `Directory` objects are implemented as separate classes, there are no special requirements to file/folder names or attributes. However, due to the difficult implementation of a directory tree, the *ls* command cannot be given a path argument as in real life, and thus can only be used in the current directory.

IV. Conclusion

In summary, this project dives into many core concepts and facets of penetration testing, and simulates them to feel like the real-world counterparts. Many of the commands that simulate complicated software are coded creatively to look realistic even though they only work in the specific instances inside the game. AILEE is a demo, and there is much opportunity to expand the game into something far more complex and realistic if enough time and energy was dedicated to doing so.

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].

[1] The `gcc` command creates a file `a.out`, which promptly throws a segmentation fault.