

# 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.

## CONTENTS

<b>I</b>	<b>Introduction</b>	1
<b>II</b>	<b>Investigation</b>	1
II-A	Planning the Project . . . . .	1
<b>III</b>	<b>Project Description</b>	1
III-A	Intro Screen . . . . .	1
III-B	Starting the Game . . . . .	1
III-C	Gameplay and Commands . . . . .	3
III-D	Events and Storyline . . . . .	4
III-E	Computers and Filesystems . . . . .	6
<b>IV</b>	<b>Conclusion</b>	7
	<b>References</b>	8

## 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:

```

1  # -*- coding: utf-8 -*-
2
3  from termcolor import colored
4  import funfunctions
5
6  import time
7  import traceback
8  import sys
9  import random
10
11 import executables
12import events
13from MainMenuException import MainMenuException
14
15DEFAULT_PROMPT = colored("AILEE@{COMP}: {CWD}$ ", '
green')
16
17CMD_NOT_FOUND_STRS = [
18    "command not found",
19    "Nope, don't know that one",
20    "This isn't Google",
21    "NOOB!",
22    "Segmentation fault (core dumped)",
23]
24
25
26
27class Shell(object):
28    """
29    Like a seashell.
30    """
31
32    def __init__(self, computer, user, agent=None,
33        cwd=None, game=None):
34        """
35        Create a shell.
36        """
37
38        self.computer = computer
39        self.user = user
40        self.agent = agent
41        self.cwd = cwd or computer.fs
42        self.prompt = DEFAULT_PROMPT
43        self.running = False
44        self._command_dictionary = {}
45        self.variables = {}
46        self.game = game
47        self.history = []
48
49        self._setup()
50
51    def _setup(self):
52        funfunctions.clear()
53        s = "Loading new shell"
54
55        print(s, end='\r')
56        i = 0
57
58        # load command dictionary
59        for module in executables.__all__:
60            self._command_dictionary.update({module:
getattr(executables, module).run})
        print(s + '.'*i, end='\r')

```

```

61        i += 1
62        time.sleep(0.1)
63        time.sleep(0.3)
64        funfunctions.clear()
65        # print(constants.title)
66
67    def _get_command_from_str(self, command_str):
68        """
69        Takes a command name, returns the executable
70        object.
71        """
72
73        if command_str == '':
74            return False
75        if command_str not in self.game.
76            allowed_commands:
77            return None
78        cmd = self._command_dictionary[command_str]
79        return cmd
80
81    def run_command(self, command, args, **kwargs):
82        """
83        Runs a command.
84
85        Input must be a runnable command that
86        accepts **kwargs.
87        """
88
89        command(
90            *args,
91            computer=self.computer,
92            cwd=self.cwd,
93            user=self.user,
94            agent=self.agent,
95            shell=self,
96            game=self.game,
97            **kwargs
98        )
99
100    def take_input(self):
101        user_input = input(self.prompt.format(
102            COMP=str(self.computer.name),
103            CWD=str(self.cwd),
104            USER=self.user),
105        )
106
107        parts = [p.strip() for p in user_input.split(
108            ' ')]
109        command = parts[0]
110        args = parts[1:]
111
112        return command, args
113
114    def one_command(self):
115        command, args = self.take_input()
116        cmd = self._get_command_from_str(command)
117        if cmd is None:
118            self.cmd_not_found()
119            return
120        elif cmd is False:
121            return # nothing on empty commands
122        cname = cmd.__module__.split('.')[1]
123        if not cname == 'doStory':
124            self.game.history.append([cname, args])
125            self.history.append([cname, args])
126
127        if not (command or args):
128            return # skip empty input
129        self.run_command(cmd, args)
130
131    def halt(self):
132        self.running = False
133
134    def cmd_not_found(self):

```

```

131     self.history.append([None, []])
132     self.game.history.append([None, []])
133     print("Command not found")
134     # print(random.choice(CMD_NOT_FOUND_STRS))
135
136     def start_shell_loop(self):
137         self.running = True
138         while self.running:
139             try:
140                 self.run_command(events.doStory.run,
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156

```

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 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.

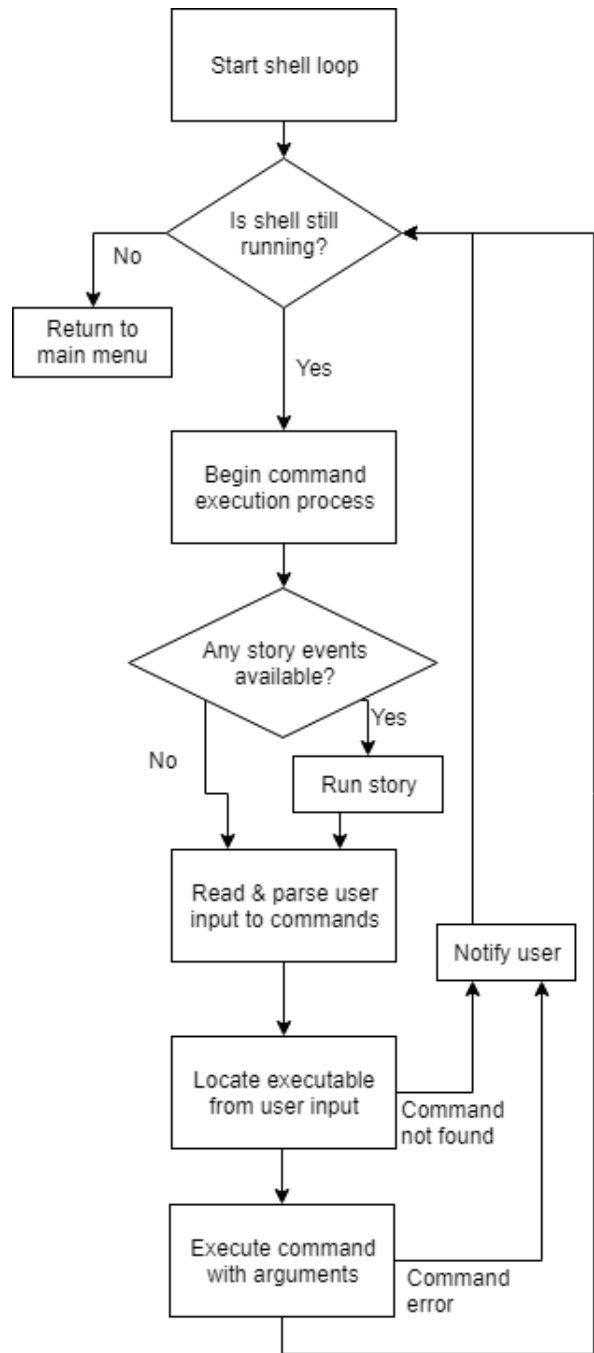


Fig. 1. Shell flow diagram

```

1  """
2  "Start the exploitation station."
3
4  Description: exploit is a software used for gaining
5  unauthorized access to remote
6  computers. There are different exploits within the
7  framework for the user to choose
8  from. Upon running the exploit software, user will
9  need to choose the exploit to
10 try, enter the target IP address, enter the port to
11 connect to, and then type
12 "run" to attempt the exploit.
13
14 Usage: exploit
15 """

```

```

12 # Would like 2 different exploit options to start
13 # with
14 # one for 'windoors' systems and one for 'lionux'
15 # systems
16
17 from funfunctions import dots
18
19 def run(*args, **kwargs):
20     emptyList = True
21     for arg in args:
22         if arg:

```

```

22     emptyList = False
23     assert len(args) == 0 or emptyList, "Invalid use
    of exploit.\n\nUsage: exploit"
24
25     print('***Welcome to the exploitation station***')
26     print('Available exploits:')
27
28     vdb = kwargs['game'].vuln_database
29     visible_exploits = [name for name in vdb if not
    name.startswith('_')]
30     exploits = {i+1: vdb[i] for i in range(len(
    visible_exploits))}
31
32     for i, exploit in exploits.items():
33         print("{:2d}. {}".format(i, exploit))
34
35     sel = 0
36     while sel not in exploits.keys():
37         try:
38             sel = int(input("Exploit selection > "))
39         except ValueError:
40             print("Enter a number")
41
42     addr = input("Enter target IP address > ")
43     port = -1
44     while port < 0:
45         try:
46             port = int(input("Select port to use > "))
47         except ValueError:
48             print("Enter a number")
49     chkrun = input("Type 'run' to begin exploitation
    > ")
50
51     if chkrun != 'run':
52         return
53
54     dots("Running exploit", 9, 0.333)
55
56     try:
57         box = kwargs['game'].network[addr]
58         vuln = exploits[sel]
59     except KeyError:
60         print("Invalid options specified.")
61         return
62
63     if (vuln in box.vulns) and \
64         (port == box.vulns[vuln][1]):
65         box.vulns[vuln][0] = True
66         print("Exploit success!")
67         kwargs['shell'].run_command(
68             kwargs['shell'].get_command_from_str('
    shell'),
69             ['new', addr]
70         )
71     else:
72         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.

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).

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).

```

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 exploitation > run
Running exploit....

```

Fig. 2. Running the *exploit* executable

```

Loading new shell.....

```

Fig. 3. Gaining a shell on target machine

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 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

```

AILEE@safeandsecurebanking: /$ 

```

Fig. 4. A new shell on an exploited target

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.

```

1 #Third dialogue of the game
2 #triggers after port scanning has been done
3 import time
4 from funfunctions import typewriter
5 from termcolor import colored
6
7 def check_run(*args, **kwargs):
8     # check for 'pscan' in history, most recently run,
9     # and exactly once
10    if len(kwargs['game'].history) == 0:
11        return False
12    if not 'event2' in kwargs['game'].events_run:
13        return False
14
15    command = ['pscan', ['120.45.30.6']]
16
17    a = kwargs['game'].history[-1] == command
18    return a
19
20 def run(*args, **kwargs):
21     # using kwargs we can get access to the shell, and
22     # from within the event
23     # have the user run commands
24
25     color = 'cyan'
26     game = kwargs['game']
27
28     text = [
29         '\nGood job, Ailee. I see you have successfully
30         found which open ports are running\non our
31         target.\n\n',
32         'Our next step is to run our vulnerability
33         scanning software against the target\nto see if
34         we can use any exploits against them.\n\n'
35     ]
36
37     filename = 'message03.txt'
38     if filename not in game.eventLogDir:
39         game.eventLogDir.addFile(filename, colored('',
40             join(text), color))
41
42     if not game.skip_dialog:
43         typewriter(colored(text[0], color))
44         typewriter(colored(text[1], color))
45     else:
46         print(colored('Event3 text skipped', 'red'))
47
48 # 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.

This is the text for the third event (Fig. 4), 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.

```

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

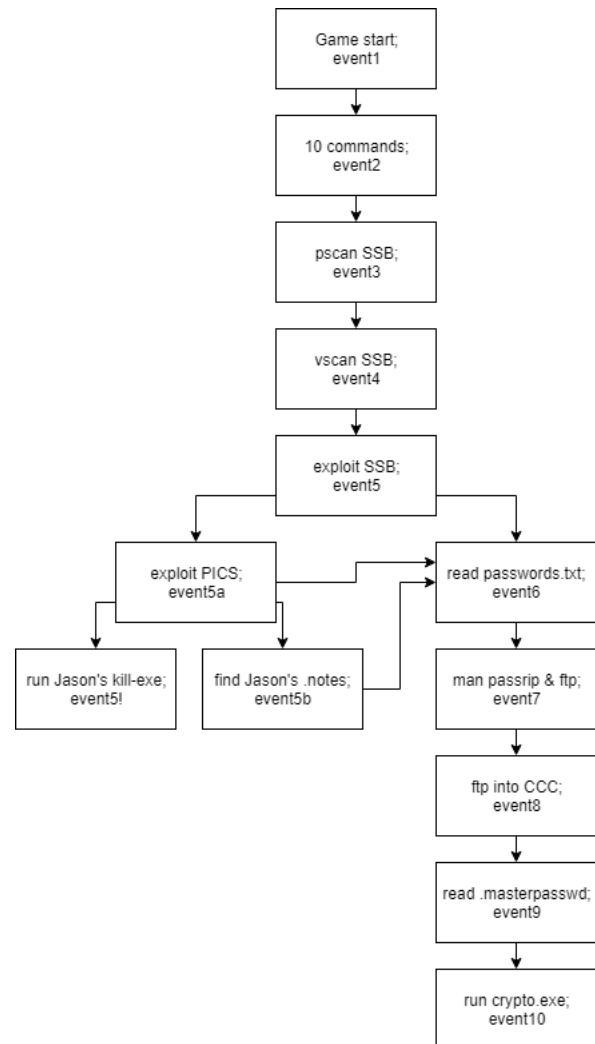


Fig. 6. The sequence of events in play order

## 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.

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.

```
1 import hashlib
2
3
4 class Permissions(object):
5     """
6     rwxrwx
7     ||||\ Owner exec
8     ||||\ Owner write
9     |||\  Owner read
10    ||\   General user exec
11    |\    General user write
12    \     General user read
13    """
14
15    def __init__(self, perms):
16        self._bits = perms
17        assert len(self._bits) == 6
18
19    def _set_bit(self, n, newval):
20        bits = list(self._bits)
21        bits[n] = newval
22        self._bits = ''.join(bits)
23
24    def __str__(self):
25        return self._bits
26
27    def __eq__(self, other):
28        if isinstance(other, Permissions):
29            return self._bits == other._bits
30        elif isinstance(other, str):
31            return self._bits == other
32        else:
33            return False
34
35    def __hash__(self):
36        return hash(self._bits)
37
38    def __getitem__(self, bit):
39        return self._bits[bit]
40
41    def __setitem__(self, bit, newval):
42        self._set_bit(bit, newval)
43
44    @property
45    def read_users(self):
46        return self._bits[0] == 'r'
47
48    @read_users.setter
49    def read_users(self, allowed):
50        self._set_bit(0, 'r' if allowed else '-')
51
52    @property
53    def write_users(self):
54        return self._bits[1] == 'w'
55
56    @write_users.setter
57    def write_users(self, allowed):
```

```
        self._set_bit(1, 'w' if allowed else '-')
58
59    @property
60    def exec_users(self):
61        return self._bits[2] == 'x'
62
63    @exec_users.setter
64    def exec_users(self, allowed):
65        self._set_bit(2, 'x' if allowed else '-')
66
67    @property
68    def read_owner(self):
69        return self._bits[3] == 'r'
70
71    @read_owner.setter
72    def read_owner(self, allowed):
73        self._set_bit(3, 'r' if allowed else '-')
74
75    @property
76    def write_owner(self):
77        return self._bits[4] == 'w'
78
79    @write_owner.setter
80    def write_owner(self, allowed):
81        self._set_bit(4, 'w' if allowed else '-')
82
83    @property
84    def exec_owner(self):
85        return self._bits[5] == 'x'
86
87    @exec_owner.setter
88    def exec_owner(self, allowed):
89        self._set_bit(5, 'x' if allowed else '-')
90
91
92 class Directory(object):
93     """
94     Tree structure of directories and files
95     """
96    def __init__(self, name=None, parent=None,
97                 children=None, permissions='r-xr-x',
98                 owner=None):
99        self.name = name or ''
100        if parent is None:
101            self.parent = self
102        else:
103            self.parent = parent
104        self.permissions = Permissions(permissions)
105        self.owner = owner or 'n/a'
106        self.children = {
107            '.': self,
108            '..': self.parent
109        }
110        if type(children) is dict:
111            self.children.update(children)
112
113    def mkdir(self, name, **kwargs):
114        assert (name not in self.children), '
115        Directory already exists'
116        newDir = Directory(name=name, parent=self,
117                           **kwargs)
118        self.children.update({name: newDir})
119        return newDir
120
121    def addFile(self, fileName, fileContents, **
122               kwargs):
123        assert (fileName not in self.children), '
124        File already exists'
125        assert ('.' in fileName), 'File has no type'
126        newFile = File(fileName, fileContents, **
127                        kwargs)
128        self.children.update({fileName: newFile})
129        return newFile
```

```

126 def addPrebuiltFile(self, file, **kwargs):
127     assert (file.name not in self.children), "
File already exists"
128     self.children.update({file.name: file})
129     return file
130
131 def rmFile(self, fileName):
132     assert (fileName in self.children), "File
does not exist"
133     assert ('.' in fileName), "File has no type"
134     del self.children[fileName]
135
136 def __iter__(self):
137     return (fName for fName in self.children)
138
139 def __repr__(self, base=True):
140     output = ""
141     if self.parent is not self:
142         upl = self.parent
143         while isinstance(upl, Directory):
144             output = upl.name + "/" + output
145             if upl is upl.parent:
146                 break
147             else:
148                 upl = upl.parent
149         output += self.name
150     if base:
151         output += '/'
152     return output
153
154 __str__ = __repr__ # set __str__ as the same
method as __repr__
155
156 def __getitem__(self, item):
157     assert (item in self.children), "File or
Directory not found"
158     return self.children[item]
159
160 def __len__(self):
161     # Remove the "." and ".." folders from the
size
162     # They're not really there
163     return len(self.children) - 2
164
165 class File(object):
166     """
167     Stores things. Like data, machine code, and
blackmail
168     """
169     def __init__(self, name, data='', permissions='r
—r—', owner=None, **kwargs):
170         self.name = name
171         self._data = data
172         self.permissions = Permissions(permissions)
173         self.owner = owner or 'n/a'
174         self._original_hash = hashlib.md5(data.
encode('utf-8')).hexdigest()
175         self._current_hash = hashlib.md5(data.encode
('utf-8')).hexdigest()
176         self._kwargs = kwargs
177
178     def append(self, data):
179         self._data += data
180         self._current_hash = hashlib.md5(self._data.
encode('utf-8')).hexdigest()
181
182     @property
183     def original_hash(self):
184         return self._original_hash
185
186     @property
187     def current_hash(self):
188         return self._current_hash
189

```

```

190 @property
191 def data(self):
192     return self._data
193
194 @data.setter
195 def data(self, newdata):
196     self._data = newdata
197     self._current_hash = hashlib.md5(self._data.
encode('utf-8')).hexdigest()
198
199 def __repr__(self):
200     return self.name
201 __str__ = __repr__ # set __str__ as the same
method as __repr__
202
203 def __len__(self):
204     return len(self._data)
205

```

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 executable and readable. While there are not yet any provisions for editing/creating files in-game<sup>1</sup>, 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 a **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

<sup>1</sup>The `gcc` command creates a file `a.out`, which promptly throws a segmentation fault.

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].