Getting Started in pwnyOS

Welcome to **pwnyOS.**

pwnyOS is a custom x86 operating system that supports link-time kASLR, multitasking and kernel threads, execution of genuine ELF files, a realtime high resolution graphics engine, and a custom hierarchical file system. This OS was written from the ground up with its use as a challenge for UIUCTF 2020 in mind. All source code in the OS is 100% custom handwritten C and assembly- there are no libraries used, and none of its code can be found anywhere online. This competition simulates an unprivileged user with physical access to a keyboard and terminal attempting to gain local privilege escalation on an unfamiliar system.

For the purposes of immersion, we are **not** providing a kernel binary- all information you need can be gleaned from the on-screen interface of the OS, and this document, just as a real attacker would have to do in our scenario.

This sequence of challenges is intended to be an increasingly difficult series of systems level exploitation. All exercises in the "kern" category can be completed with only printable ASCII characters typed by hand into a VNC window. The exploits here are intended to provide insight into core systems concepts, and are categorized by the systems concept they are testing. For our newcomers, a comprehensive guide explaining the basic prerequisite knowledge will be provided as necessary. Our intention is that everything you need to complete these challenges is provided to you- no guessing is involved. Regardless of your level of experience, an understanding of the concepts from this document is necessary to solve several of the challenges, and as in real bug finding, solving these challenges will also require a degree of intuition.

These challenges are neither strictly PWN nor RE, hence this series gets its own unique category. The common theme is every challenge in this sequence is centered around a particular systems concept. No challenge should involve any pure guessing, but these challenges do require some intuition and a familiarity with systems concepts. This guide, along with the syscalls guide, contains everything you need to know to completely exploit pwnyOS. Don't be afraid to get your hands dirty searching for bugs- there are plenty of them just waiting to be discovered.

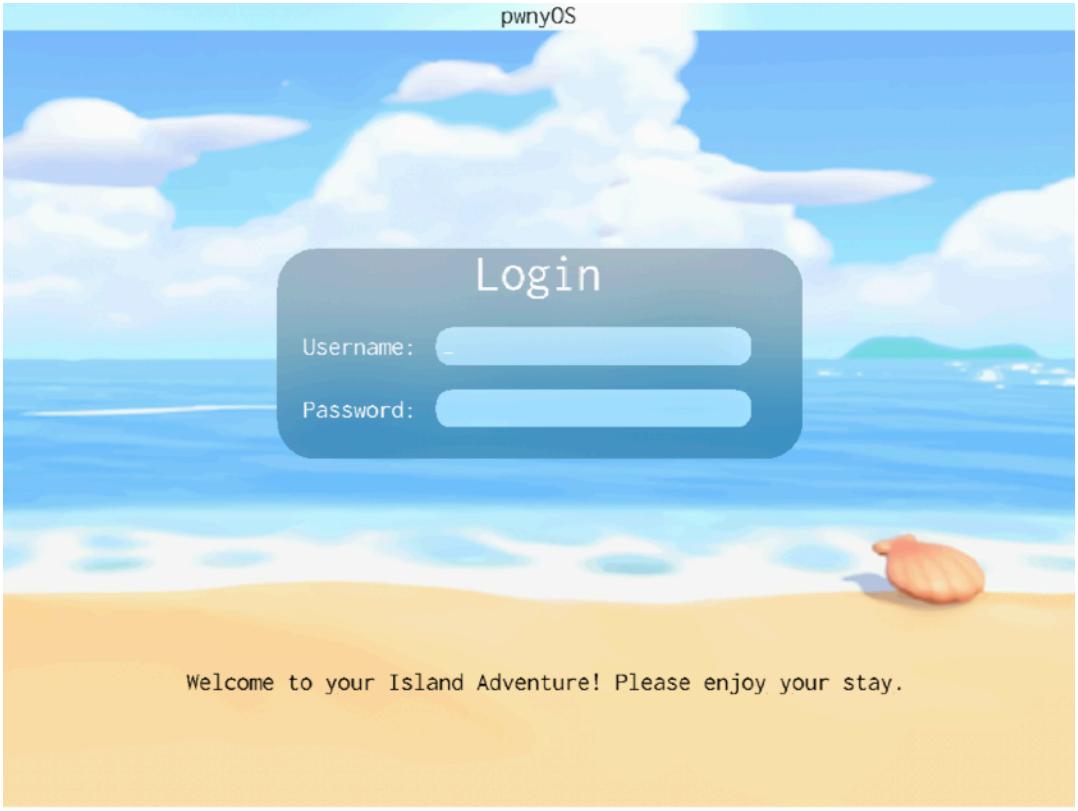
Welcome to pwnyOS.

~ravi

Time Limits

After a period of 15 minutes of inactivity, your VM will automatically restart, and you may loose your progress. Make sure to keep a record of your exploits and any flags you receive, so you can quickly get back to where you started should you need to take a break.

Login Screen



The Login Window is the first thing you see when you startup pwnyOS. pwnyOS is a keyboard-only interface, and all actions can be performed from the keyboard alone. That is, all exploitation can be done using no more than printable ASCII characters typed from a keyboard.

Type in a username and press enter to type in a password. After typing your password, you'll either see a notification box indicating your username or password was invalid, or you'll be dropped into the first sandbox. If you got an error, just press enter to dismiss it.

The **Sandbox**

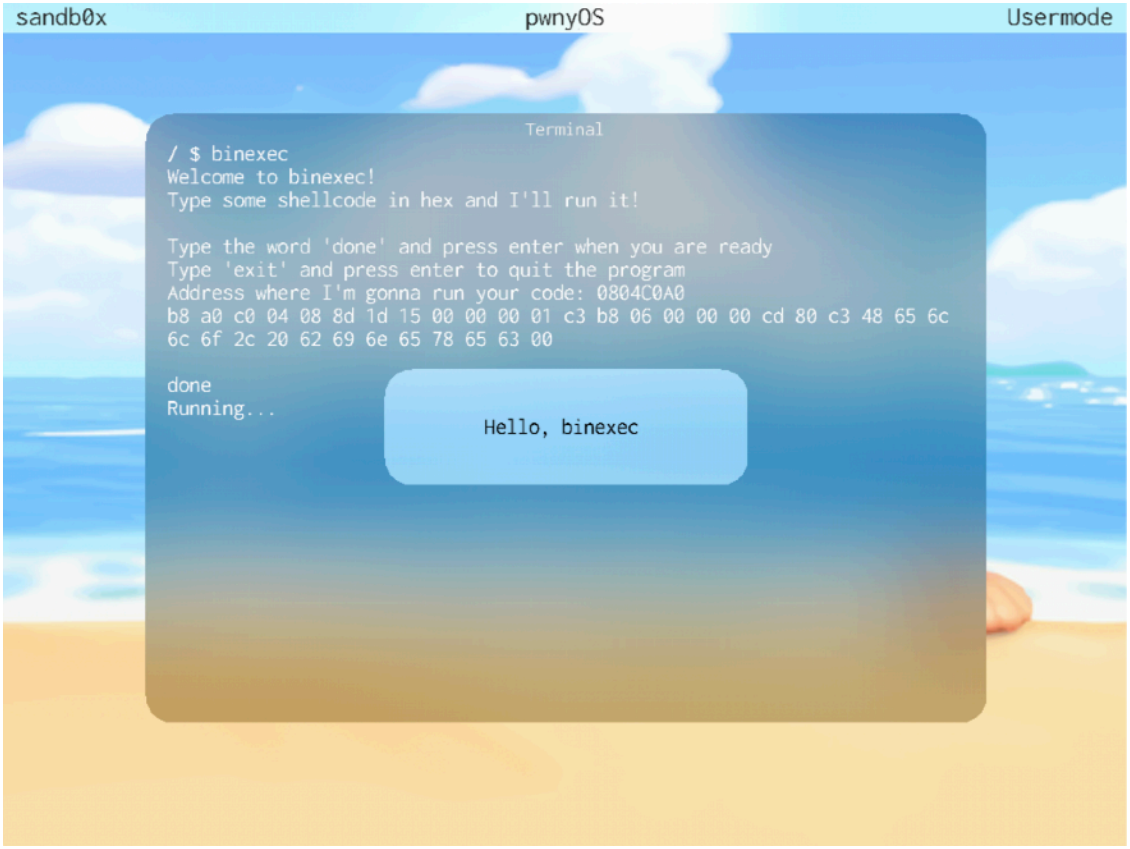
Once you've logged in as **`sandb0x`**, you'll be presented with some more info about the next challenge. After pressing enter, you'll be dropped into "**`binexec`**."

`binexec` is a program that will run whatever shellcode you provide. However, you're still in a sandbox, so many of the system calls in pwnyOS won't work yet.

To break out of the binexec loop, write and run some shellcode that calls the "SANDBOX_SPECIAL" syscall. The argument you give it is irrelevant here.

For more information about syscalls, check out the provided system call documentation.

Binexec



This is the binexec interface. Binexec allows you to type in i386 bytecode as ASCII characters, and will run your code when you type “done” and press enter.

In this example, the following bytecode is being run:

```
movl $0x0804c0a0, %eax
leal str, %ebx
addl %eax, %ebx
movl $6, %eax
int $0x80
ret

str:
.string "Hello, binexec"
```

Keep in mind when you are inside of a sandbox, you may not be allowed to use certain syscalls. Syscall 6 used here (SYS_ALERT) may be disallowed. Also, keep in mind the address binexec chooses to run your code at may be different from this address as well.

Rash

Once you've escaped the binexec loop, you'll be dropped into the pwnyOS official shell, RASH. Navigating rash is quite simple.

To list all commands, type "help":

RASH **Commands**

ls: List all files in current directory

cd: Change directory to a given directory

exit: Quit RASH

alert: Display a popup with a message

su / su (username): Switch user. Defaults to root if no name provided.

whoami: Display current user information

motd: Show the message of the day

lsproc: List all processes

help: Display the help menu

reboot: Attempt to reboot the system

shutdown: Attempt to shutdown the system

pwd: Print working directory

cat file: Print a file

run file: Execute a file

To run a program, you can also just type its name. For example, to run another instance of rash, type:

```
/bin/rash
```

Files in /bin can be executed without typing their absolute path. So, you could also run rash like this from any directory:

```
rash
```

pwnyOS **Executables**

pwnyOS programs are standard ELF files, except they are limited in a few ways.

1. They must be statically linked
2. They must be position dependent, and start at **`0x08040000`**
3. They must place all initialized data in the .text segment, not the .data segment

Additionally, pwnyOS extends the definition of a standard ELF file to support privileged binaries. Namely, the first 4 bytes of the ELF header determine the permission level of the program.

Standard ELF headers begin with the byte sequence:

```
'F' 'L' 'E' 0x7F
```

On pwnyOS, this file would be run under the same permissions as the calling process.

pwnyOS supports binaries that are allowed to run as a more privileged user, using the last byte of the ELF header to specify user ID.

The following byte sequence would cause a program to run with user ID 2:

```
'F' 'L' 'E' 0x82
```

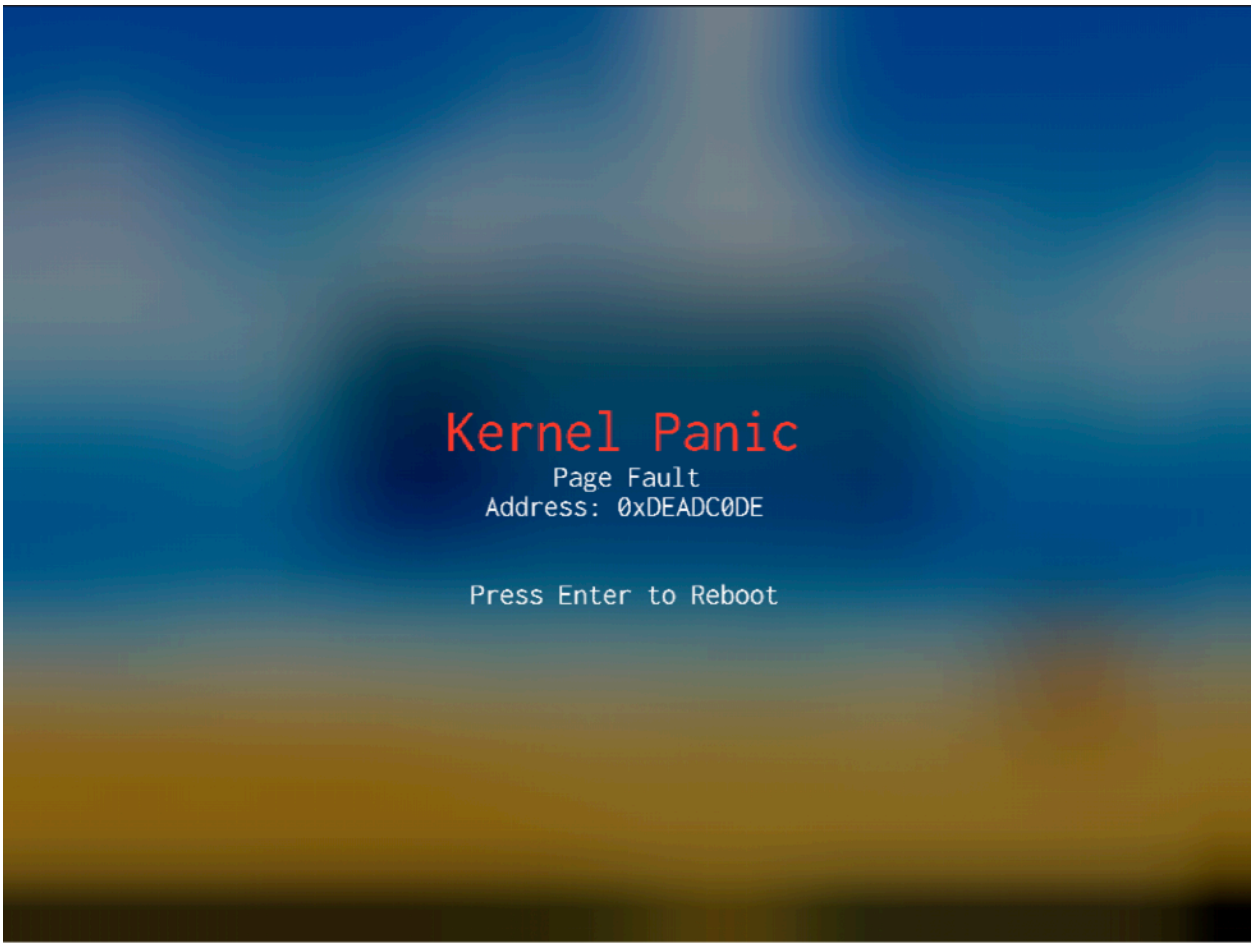
The lower 7 bits of the last byte define the user ID to run the program as, and the most significant bit of the last byte selects the mode the new program runs in.

If the most significant bit is not set, the program will be run in non-blocking mode. That is, it will run concurrent to the currently running program, instead of causing the current program to be blocked until the new program completes.

The following byte sequence would launch a non-blocking program with user ID 2:

```
'F' 'L' 'E' 0x02
```

Kernel **Panics**



In some rare cases, pwnyOS may encounter an exception that it cannot recover from. If that happens, a “kernel panic” occurs. A kernel panic generally means a processor exception occurred while running kernel code, or the kernel encountered a known sequence of code that causes it to crash itself.

In pwnyOS, kernel panics may arise from processor exceptions, or from kernel code self-detecting an error, and preemptively crashing.

If you see a kernel panic, just press “enter” to reboot. The error code presented may be beneficial for your exploitation, as it gives an insight into why the given exception occurred.

In this example, a “page fault” hardware exception was generated in the kernel when it attempted to read or write from address **`0xDEADC0DE`**. Since this is not a valid address, the kernel crashed.

pwnyOS **Filesystem**

The pwnyOS filesystem has a few folders to take note of.

`/bin` contains all binaries, including binexec and rash.

`/prot` is a protected directory, containing valuable secrets. You need root user privilege to access this resource!

`/sandb0x` contains all the files associated with the **`sandb0x`** account.

`/user` contains all the files associated with the **`user`** account.

`/root` contains all the files associated with the **`root`** account.

The pwnyOS filesystem is **read-only**: if you attempt to write to a pwnyOS filesystem file descriptor, nothing will happen! You also cannot create new files.

pwnyOS also supports a special folder called **`/proc`**. This file is not visible from the shell, as it lives outside of the filesystem the shell directory system can operate on. However, files under **`/proc`** can be opened and read from by the shell. There is only 1 file, called "all," that lists all processes running on the system. To read from it, use:

```
/proc/all
```

The **`lsproc`** command provides a convenient program that just opens **`/proc/all`** and reads from it, and is the recommended way of reading all processes.

All files in pwnyOS have a purpose- if you see a file, it is there for a reason.

pwnyOS **Keyboard Shortcuts**

Control + L: Clear screen.
Control + X: Reboot VM. (In case of an unrecoverable error)

pwnyOS **Behaviors to Know**

If you remain inactive for more than 15 minutes, your VM will automatically reboot to save resources. This may cause you to lose your progress, so make sure to write down flags, and save your exploits in case you need to run them again!

About the **Author**

This document was created accompanying the operating system “pwnyOS” written for UIUCTF 2020 presented by SIGPwny. It was written by ravi (jprx).