# PwnyOS 1.1 Release Notes

Last summer, we released PwnyOS 1.0: a rock-solid platform on which we have built this new release. PwnyOS 1.0 introduced support for our custom filesystem (PwnyFS), preemptive multitasking, numerous process management syscalls, and scheduling primitives. Our users loved the experience provided by PwnyOS 1.0, and this year, we're taking the PwnyOS experience to the next level.

This summer's release, PwnyOS 1.1, is the clearest expression yet of our vision for the future of PwnyOS. We listened carefully to our users' feedback, and we think our users will love the changes provided by the new system.

This document contains all the changes we've made to the kernel, as well as a guide to help you solve the challenges provided by the new release.

And as usual, there are plenty of bugs just waiting to be discovered. Welcome back to PwnyOS.

---

## Disclaimer and License

PwnyOS is an operating system designed to be insecure, buggy, and vulnerable. Throughout this challenge, you may gain access to the PwnyOS ISO image and kernel binary. Booting this operating system may result in **permanent damage** to your computer, motherboard, CPU, or firmware, or may result in permanent data loss. The authors take no responsibility for any damages caused by the software. The intention is that the PwnyOS image is analyzed statically or run under an emulator- never booted directly onto bare hardware. The authors assume no responsibility for any damages caused by the software, even when running the OS under an emulator. Users should only run the OS at their own risk.

The PwnyOS binary is released under the MIT License. See the License section at the end of this document for more information.

# So, what's new with PwnyOS?

This year's release, PwnyOS 1.1, is an incremental release that builds on the code base of last summer's release (1.0). This document is a companion document that builds on the documentation provided with PwnyOS 1.0, which can be found here: https://github.com/sigpwny/pwnyOS-2020-docs.

The top three requested PwnyOS features by UIUCTF 2020 participants were:

1. More PwnyOS instances
2. A serial console for interfacing with **`pwntools`**
3. Access to the PwnyOS kernel image to reverse engineer

This year, we're happy to announce we are introducing all three of these, as well as some exciting new kernel features.

And of course, we are also introducing our most difficult PwnyOS challenges ever.

## Serial IO

For PwnyOS 1.1, we are returning to the basics with a new RS232 serial IO device. The high resolution GUI has been replaced with simple serial IO, allowing users to type non-printable characters into the console and script their exploits. There is no support for backspace with this interface, as the serial backspace character **`0x7F`** tends to be quite useful for writing shellcode. As a result, we've made the decision to not support backspacing at all to hopefully simplify your shellcode writing process. There are three characters to avoid in any shellcode that will be treated as special signals, however.

The characters **`0x0A`** and **`0x0D`** each act as a line submission character. Anytime either is observed in the input buffer, the buffer will be sent to the consuming reader. So, if either character is present midway through some shellcode, your overflow will stop there.

The character **`0x02`**, or ASCII start of text, acts as the break character in PwnyOS. We opted to use ^B instead of ^C to simplify interfacing with the serial terminal. If this character is included in shellcode, the process will immediately fire **`SIGINT`** upon the assertion of this character.

## Signals

We have added support for user-mode interrupts (“signals”). Users can register an asynchronous callback method that executes when some particular action occurs. There are two signal handlers supported in PwnyOS 1.1, which are:

| Number | Meaning |
|---|---|
| `0` | `SIGSEGV` |
| `1` | `SIGINT` |

`SIGSEGV` is fired whenever the process causes a page fault or general protection fault. `SIGINT` is fired whenever the user presses the “break” keyboard combination, which is ^B (control + B), or ASCII `0x02`.

If a process doesn’t install a signal handler, then the signal event will cause the process to exit just like before. The signal management syscalls and internals are described later on in this document.

# Image Layout

**Warning**: Run the PwnyOS ISO at your own risk! Running on real hardware is not supported.

The PwnyOS ISO is an ISO 9660 GRUB filesystem containing the kernel and PwnyFS filesystem image. This image was created with the following command:

```
grub-mkrescue ./files -o PwnyOS.iso > /dev/null 2>&1
```

**`grub-mkrescue`** is a command line utility for producing GRUB bootable rescue images with a given kernel. In this case, the kernel is the PwnyOS kernel, which is a freestanding 32-bit ELF binary. The PwnyFS filesystem image is also included in the files directory, which is loaded by GRUB as a multiboot boot module and later read by the PwnyOS kernel. To extract the files from the ISO image, the 7zip command line utility can be used as follows:

```
7z x PwnyOS.iso
```

Be careful, as this will place the contents of the ISO in your current directory!
The following folders should be present on the ISO:

```
[BOOT]   boot   boot.catalog   fs   main.pwnyfs
```

The kernel ELF itself can be found at **`boot/PwnyOS`**.

The PwnyFS filesystem image can be found at **`main.pwnyfs`**. The **`fs`** directory provides a copy of the contents placed into the PwnyFS filesystem on the ISO 9660 filesystem so you can browse them without having to unpack the PwnyFS filesystem.

The GRUB launch script can be found at **`boot/grub/grub.cfg`**. This script defines the PwnyOS kernel as a multiboot compliant kernel ELF, defines the PwnyFS filesystem as a boot module, and launches the kernel.

The **`[BOOT]`** directory, **`boot.catalog`**, and most of the files in the **`boot`** directory are automatically generated by GRUB and are not related to PwnyOS.

# A Brief Aside:
# The x86 Calling Convention

This year, we wanted to provide a bit more background information on the x86 architecture, specifically the aspects unique to operating systems. We hope that this helps beginners who are new to this kind of challenge get started exploring systems concepts. Low level programming and reverse engineering can seem quite intimidating, but there is nothing to be afraid of :)

How are functions called? At the assembly level, functions (AKA methods or subroutines) are just chunks of instructions that can be called by other code. Since a function can be called from anywhere, we need a way to keep track of who called a given function, so we can return back to the caller once we're done. The **caller** is the chunk of code executing a function, and the **callee** is the function we are executing.

The **stack** is a region of memory used by functions to keep track of the caller's state. This includes the instruction that the caller should execute when this function completes (the "return address"), saved registers (on 32-bit you'll see **`ebp`** saved at the start of pretty much every function), and other things such as arguments to the called function. Stacks follow the "last in, first out" scheme- the first thing pushed to the stack is the last thing popped off of it.

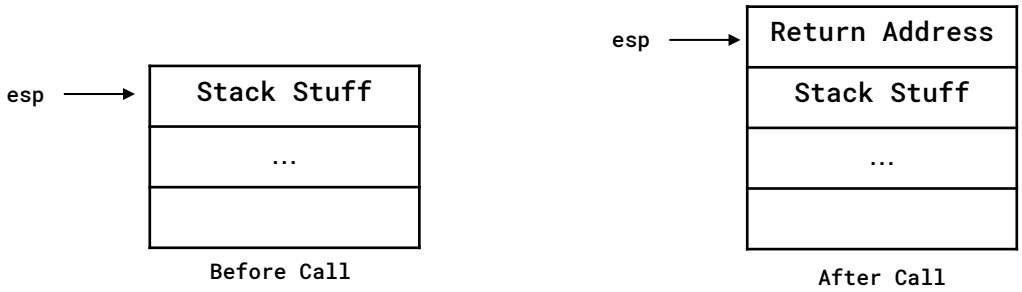
x86 has 8 main program registers, a flags register, an instruction pointer, and 6 segmentation registers. The main registers are **`eax, ebx, ecx, edx, esi, edi, esp,`** and **`ebp`**. There is a lot of history to the naming convention and purpose of each register, but for the sake of this introduction, you can treat them all (except **`esp`**) as being free for use by our program. **`esp`** is the stack pointer and always points to the top of the stack. It is automatically changed whenever any instruction that modifies the stack is executed (such as **`push`**, **`pop`**, **`call`**, or **`ret`**). **`eax`** is where we save the return value that is passed from the callee to the caller when the callee returns.

The flags register, **`eflags`**, contains many bit flags that correspond to various parts of the processor state. This can be anything from the result of arithmetic operations to various system state flags. For more information on the eflags bits, check out the Wikipedia page: https://en.wikipedia.org/wiki/FLAGS_register.

The instruction pointer, **`eip`**, points to the instruction currently being executed.

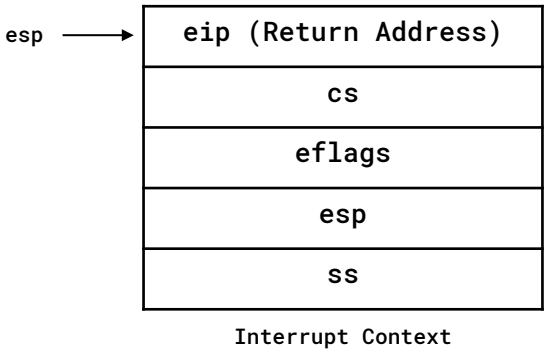
The segment selectors are **`cs`**, **`ds`**, **`es`**, **`fs`**, **`gs`**, and **`ss`**. They are vestigial remnants of a very cool feature of early Intel processors called segmentation. However, in modern operating systems, segmentation protections have been superseded by paging, and on PwnyOS segmentation is not used (so the contents of these registers can be ignored).

The **`call`** instruction is the main way to call a function on x86. Call will save the next instruction to execute to the stack, and then change the instruction pointer to point at the function to execute. Since we recorded the next instruction after the call on the stack, the callee can find where to return to when it's done.




The **`ret`** instruction undoes the **`call`** instruction by popping the return address off the stack, and setting **`eip`** to point to that location. This allows the callee to pass execution back to the caller with the return value saved in **`eax`**.

**Interrupts** are asynchronous control transfers that can occur at any time. PwnyOS receives many kinds of interrupts. For example, when you press a key on the emulated PS/2 keyboard, PwnyOS will receive an interrupt informing it that a key was pressed. Interrupts can happen at any time, so the interrupt routine must be careful to exit with the registers containing the same values as they had when it began. Additionally, the processor needs to keep track of more than just the return address during these kinds of control transfers. When an interrupt occurs, the following structure is pushed to the stack:



The interrupt context structure contains the return address, just like with the call instruction. However, we also save the **`cs`** and **`ss`** segment registers, **`eflags`**, and **`esp`**. The **`cs`** selector is an index into a special structure called the **Global Descriptor Table**, and defines the processor's current permission level. Level 0 (or "ring 0") is kernel mode, which has ultimate power over the system, and level 3 (or "ring 3") is user mode, which is restricted in what kinds of code it can execute. Even programs that run as root still run in ring 3 mode- ring 0 is reserved for kernel code (system calls, interrupts, exception handlers, boot code, task switching code, and IO are all examples of the kinds of things that run in ring 0). While segmentation is unused in PwnyOS, we still need to use the **`cs`** selector to set the processor permission level.

Interrupts need to save the previous permission level because all interrupts run in kernel mode- when the processor exits the interrupt, it needs to know which mode to switch back to. There is some nuance to interrupt control transfers omitted from this document, for more information check out the AMD Architecture Programmer's Manual Volume 2: System Programming (https://www.amd.com/system/files/TechDocs/24593.pdf).

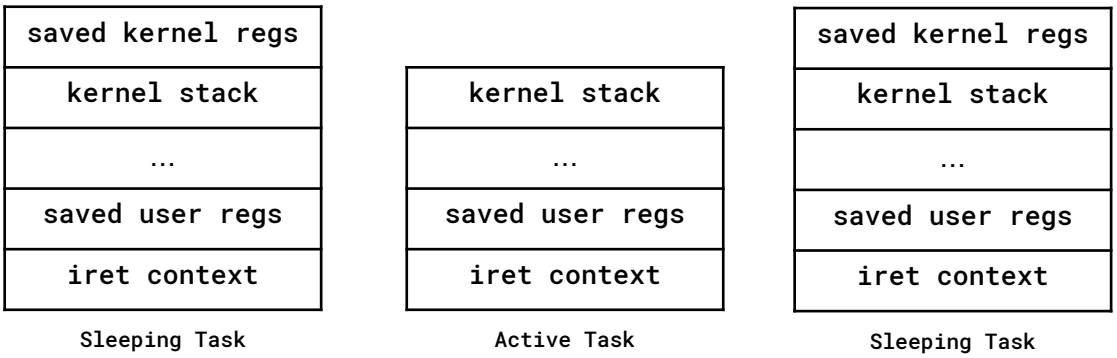
The **`iret`** instruction undoes the changes created by an interrupt. It pops the return address off the stack, sets the stack pointer, **`cs`**, and **`ss`** selectors back to what they were, and restores **`eflags`** to what it was at the interrupt time. This process assumes the interrupt routine saved any registers that it modified (which PwnyOS does- more on this in the "Signals" section). This allows the processor to return exactly back to the code it was executing prior to the interrupt.

More information on interrupts and task switching can be found in last year's "Getting Started with PwnyOS" guide, available at: https://github.com/sigpwny/pwnyOS-2020-docs.

One last question that remains is how does an operating system allow multiple tasks to run on one CPU core? The answer is the **context switch**. There are many ways to approach this, however, this document will describe briefly the approach taken by PwnyOS.

For each process in the system, the kernel allocates a 4 MB page for the user code and stack. This page is set to RWX for ring 3 execution, and the user process can do with it whatever it likes. The kernel will load the ELF program to be executed into this memory region and jump to the ELF entrypoint, and after that, the user program can do whatever it likes with its memory. The kernel

also allocates a “kernel stack” per process, which is a separate region of memory that’s used as the kernel stack during kernel code execution. When the kernel decides a given task needs to be swapped to a different task, the kernel pushes its state to the current kernel task, and then swaps to a different task’s stack and pops the state off that stack.




Pictured above are the kernel stacks for three example processes. Note that the base of each stack is an **`iret`** context (the struct pushed during interrupt entry). This is because any time kernel code is executing, we entered the kernel via an asynchronous control transfer of some kind (system call, interrupt, or exception), and on 32-bit x86, all of these control transfers push an **`iret`** context to the stack. Just above the iret context is the saved user registers for a given task- these are pushed by PwnyOS at the entrypoint to each interrupt service routine.

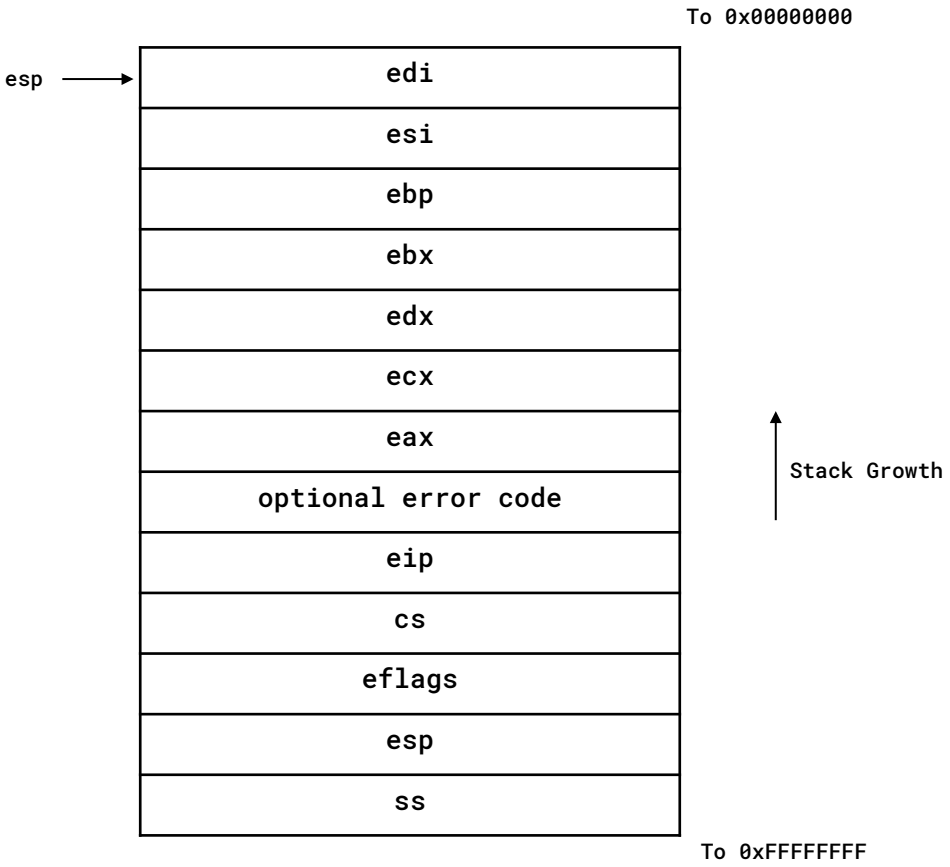
The top of a sleeping task contains all the kernel registers saved immediately before task switching. These are the callee saved registers that we need to restore before returning to previous kernel code. Below the saved registers is the kernel call stack- this is just a regular stack that we will return to when we switch back to this task. The stack and base pointers for sleeping tasks are saved by the kernel so it knows where the stack for a given sleeping process resides.

For the active task, when executing in kernel mode, the kernel stack is used as the processor stack (in **`esp`**). When the kernel eventually reaches the bottom of its stack, it will perform an **`iret`** to return to user mode by consuming the **`iret`** context at the base of the kernel stack. Active tasks running in user mode have empty kernel stacks on PwnyOS. Tasks pass a return value back to their parent task by switching to its stack and setting **`eax`** to the return value on exit.

There are many more details to context switching beyond the scope of this introduction, but hopefully this brief introduction was useful to someone!

# Signals

Now that we've taken a look at an overview of interrupt handling and context switching, how does PwnyOS handle user-mode interrupts (signals)? At the start of every asynchronous control transfer (syscalls, exceptions, interrupts), PwnyOS pushes all the registers in the following configuration:




The bottom 5 registers are actually just the interrupt context pushed by the processor. The error code is sometimes pushed by exception handlers, and if it wasn't pushed, PwnyOS will just push an empty value there. The rest of the registers are saved right on the stack (except for esp, which was already saved as part of the interrupt context).

Every kernel method inspects the saved registers, and if **`cs`** indicates we came from usermode, the kernel will store a pointer to this structure as the saved user register struct. This is used for a variety of things in the kernel, and most notably is used for signal delivery.

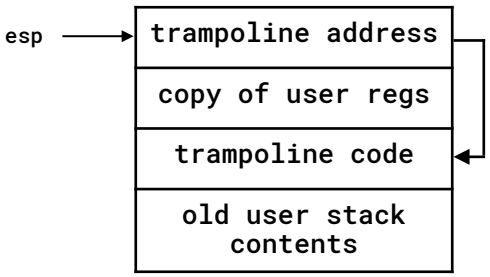
When a signal is detected, the kernel needs to modify the user registers such that the next instruction executed is the first instruction of the registered signal handler. The kernel also needs to provide the user program a means of restoring all its current registers as they are back to the current instruction. The following pseudocode describes how the kernel delivers a signal:

```
allocate SIGNAL_TRAMPOLINE_LEN bytes on user stack
      memcpy SIGNAL_TRAMPOLINE to user stack
         let trampoline_addr = user stack

 allocate sizeof(user_regs_t) bytes on user stack
          memcpy user_regs to user stack

   allocate sizeof(uint32_t) bytes on user stack
       memcpy trampoline_addr to user stack
```

First, the kernel allocates some space for a bit of linkage trampoline code on the stack. This trampoline performs the following:

```
mov eax, SYS_SIGRET
int 0x80
```

Next, the kernel pushes a copy of the entire register struct (from **edi** down to **ss**) onto the stack just above the trampoline. Then, the kernel pushes the address of the trampoline to the user stack. This leaves the user stack in the following state:




Finally, the user regs struct is updated to point **eip** at the appropriate signal handler, the general purpose registers are zeroed, and **esp** is updated to reflect the new stack. When the signal handler performs **ret**, it will jump to the trampoline and execute the **sigreturn** syscall with **esp** pointing to the copy of the user regs from immediately before the signal was executed.

The **`sigreturn`** syscall reads a saved user register struct from the top of the user stack and restores all the user registers to the values contained in that struct. For security purposes, not all registers are read.

The following registers are restored: **`edi`**, **`esi`**, **`ebp`**, **`ebx`**, **`edx`**, **`ecx`**, **`eax`**, **`eip`**, **`esp`**, **`eflags`**. The following registers are ignored: the error code, **`cs`**, **`ss`**.

The **`sigreturn`** and **`sighandler_install`** syscalls are described in more detail in the updated syscalls section of this document.

# Updated Syscalls

All of the system calls described in the documentation for PwnyOS 1.0 are still present in PwnyOS 1.1, with the sole exception of the **`SANDBOX_SPECIAL`** syscall. Additionally, all sandboxing restrictions have been lifted, as the PwnyOS 1.1 challenges give you root to start with.

| Name | Number (eax) | Arg 1 (ebx) | Arg 2 (ecx) | Return Value |
|---|---|---|---|---|
| **`sighandler_install`** | 14 | handler index | handler address | 0 on success, -1 on failure |
| **`sigreturn`** | 15 | - | - | Doesn't return on success, -1 on failure |
| **`getphys`** | 16 | - | - | Physical address of the mmap page |

### `sighandler_install`

This system call allows the user process to register a handler with the kernel to execute when a signal is detected. The handler address is verified to be a valid user address. When the signal is fired the user stack will be modified to include a linkage trampoline that calls **`sigreturn`**, the saved user registers at signal dispatch time, and a return address pointing to the trampoline.

### `sigreturn`

Reads a user register context from the user stack and restores all the registers (except for **`ss`** and **`cs`**) to the values contained on the stack. This undoes the effects of a signal.

### `getphys`

Returns the physical address of the mmap page, if it was allocated.

# Port IO

x86 architectures support two flavors of IO: memory-mapped and port IO. Memory mapped IO allows a kernel to communicate with devices by reading and writing memory. Port based IO allows a kernel to communicate with devices using special **`in`** and **`out`** instructions.

More information on port IO can be found here: https://wiki.osdev.org/I/O_Ports.

The primary ATA PIO hard drive controller is located from IO ports **`0x1F0`** to **`0x1F7`**.

# Standard IO

PwnyOS standard IO uses one file descriptor for both reads and writes- file descriptor 0. This is unlike the Linux behavior using file descriptors 0, 1, and 2 for standard IO.

So, to read from the terminal in shellcode, call read with FD = 0. To write to the terminal, call write with FD = 0.

Standard IO is buffered, and when using the serial connection, the buffer limits all input to 256 bytes. Buffered input is flushed every time the return key is pressed.

# Paging

x86 provides two different methods for memory permission restriction: segmentation and paging. PwnyOS uses paging, which is the choice many modern operating systems make. Paging partitions the entire address space into a flat sequence of blocks called “pages.” In short, all of memory is partitioned into 4MB sections, which can themselves be partitioned into 4KB sections with various permissions. The layout of pages is controlled by a structure called the page directory, which itself is comprised of page tables or huge pages. More specific information on paging in 32-bit x86 can be found at: https://wiki.osdev.org/Paging.

Additionally, more details can be found in the AMD Architecture Programmer’s Manual Volume 2: System Programming (https://www.amd.com/system/files/TechDocs/24593.pdf) Section 5.

The special control register **`cr3`** contains the address of the page directory table as a physical address. Writing a physical address to **`cr3`** will cause the Translation Lookaside Buffer (TLB) to be flushed, and update the system page table mappings accordingly. Ring 0 privileges are required to write a new page directory to **`cr3`**.

# About the Authors

This document was written for UIUCTF 2021 by ravi (jprx- https://github.com/jprx). For more info on PwnyOS, check out the PwnyOS website at:

**www.pwnyos.com**

Thanks to YiFei (zhuyifei1999), Ankur (arxenix), Kuilin (kuilin), Chris (2much4u), Ian (ian5v), and the rest of the SIGPwny team for help with the testing and deployment of the PwnyOS VMs for UIUCTF 2021.

# License