System Calls in pwnyOS

Syscall Number (eax)	Name	Arg 1 (ebx)	Arg 2 (ecx)	Arg 3 (edx)	Returns Value (always int32_t)
0	SYSRET	uint32_t ret_code	-	-	Never returns
1	EXEC	char *filename_to_exec	-	-	Process exit code
2	OPEN	char *filename_to_open	-	-	File descriptor, or negative error code
3	CLOSE	int32_t file_descriptor	-	-	0 on success or negative error code
4	READ	int32_t file_descriptor	char *read_buffer	uint32_t bytes_to_read	Bytes read
5	WRITE	int32_t file_descriptor	char *write_buffer	uint32_t bytes_to_read	Bytes written
6	ALERT	char *message	-	-	Always 0
7	ENV_CONFIG	uint32_t option_code	uint32_t option_value	-	0 on success, -1 on failure
8	REBOOT	-	-	-	-1 on permission denied
9	SHUTDOWN	-	-	-	-1 on permission denied
10	SWITCH_USER	char *username	char *password	-	UID of new user, negative on failure
11	GET_USER	char *name	size_t name_size	uid_t *id	0 on success, -1 on failure
12	REMOTE_ SETUSER	pid_t remote_proc_id	-	-	0 on success, -1 on failure
13	MMAP	-	-	-	Address of a reserved 4MB page on success, 0 on failure
14	SANDBOX_ SPECIAL	uint32_t kern_slide	-	-	Undefined

Table 1: All syscalls on pwnyOS

0: SYSRET(uint32_t retcode)

Exit a process, permanently.

1: EXEC(char *filename)

Start a new process- this call won't exit until the called process calls SYSRET! This syscall returns the exit code the called process sent to SYSRET. Arguments are specified by a space-separated list after the filename.

This uses the kernel 4MB page allocator to request a new page for the process. This page may contain program data from previous programs, or from other places the kernel uses the allocator. The new program is read into memory, overwriting the old data. However, if an old program was larger than the new one, remnants of its memory may still be present in the page! (You can't assume all memory is initialized to 0).

Returns -1 on failure (cannot execute program).

2: OPEN(char *filename)

Try to open a file. Returns a file descriptor (non-negative integer) on success, or negative error code on failure.

3: CLOSE(int32_t fd)

Close a file descriptor that was opened with open.

4: READ(int32_t fd, char *buf, size_t buf_size)

Reads buf_size bytes from the file descriptor in question. This is a seeking operation, so be sure to save the results- the next time you call read, you will be offset by buf_size in the file! Returns the number of bytes successfully read.

5: WRITE(int32_t fd, char *buf, size_t buf_size)

Same deal as read, but now you're writing to a file descriptor. This syscall is also seeking. Returns the number of bytes successfully written. (NOTE: The filesystem on pwnyOS is read-only, so this syscall is only useful for writing to stdio (fd 0) and other non-filesystem file descriptors).

6: ALERT(char *message)

Displays a nicely formatted popup on the screen to the user. RASH, the shell provided with pwnyOS, uses this internally when you use the "alert" command.

7: ENV_CONFIG(uint32_t option_code, uint32_t option_value)

Set a special environment variable indexed by option_code to the value provided in option_value. Returns 0 on success, -1 on failure. This typically fails if the option_code doesn't exist.

8: REBOOT()

Reboot the OS. Returns -1 if the user doesn't have permission to do so.

9: SHUTDOWN()

Shutdown the OS. Returns -1 if the user doesn't have permission to do so.

10: SWITCH_USER(char *username, char *password)

Attempt to login as a different user. This will switch the UID associated with the currently executing process. Returns the new UID on success, or a negative error code on failure.

11: GET_USER(char *name, size_t name_size, uid_t *id)

Get information about the current user. This will write the username into the name field, and will set the id pointer to the user's UID. Returns 0 on success, -1 on failure.

12: REMOTE_SETUSER(pid_t remote_proc_id)

Set a remote process's permission level to the calling process's permission level. If the remote process is of equal or greater privilege, then this does nothing. Remote Process ID is a pid_t, which is just the integer value of the process ID.

13: MMAP()

Request a 4 MB page of memory from the pwnyOS page allocator. Each process can only request 1 mmap- every future call to mmap will return NULL after the first one. pwnyOS always maps mmap to the same virtual address, **`0x0D048000`**. After the call to MMAP, 4 MB after this address are readable, writeable, and executable by the user process that called MMAP. There is no way for a user process to free this page- it is automatically returned to the page allocator on SYSRET.

14: SANDBOX_SPECIAL(uint32_t kern_slide)

A special system call used for the first few challenges in pwnyOS. Call this system call with any argument to escape the first binexec loop. Later, if you find the kASLR base address, call this syscall with the slide value in ebx and you'll be awarded with a flag.

Sandboxing Policy

When you first enter pwnyOS, you'll be placed into a looping call to binexec. While you're stuck in this loop, the following syscalls are the only allowed ones:

SYSRET
OPEN
CLOSE
READ
WRITE
ENVCONFIG
SANDBOX_SPECIAL

After exiting the binexec call by calling the SANDBOX_SPECIAL syscall, a few more syscalls will be allowed, but not all. The following syscalls will still be denied:

SWITCH_USER
GET_USER
MMAP
REMOTE_SWITCHUSER
REBOOT
SHUTDOWN

Once you login to the "user" account, all sandboxing system call restrictions are lifted. However, the kernel can still enforce that arguments are valid, and may still deny system calls if it detects invalid arguments.

For example, the account named "user" lacks the entitlement to reboot or shutdown the system, so these system calls still fail.

What's a System Call?

Any time a process needs to interact with something outside of its own memory space or the processor registers, it needs to ask the OS to perform some task on its behalf. The things an OS can do for a user process are called **system calls**, or "syscalls" for short. These aren't regular functions, although they behave similarly to them from a programming point of view.

For example, here's a list of the system calls exposed by Linux:
https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md

To perform a syscall on 32-bit Linux (and on pwnyOS), arguments are passed via registers instead of the stack (if you're curious why, see footnote 1), and then a special instruction is called. This special instruction is called **INT**, or "interrupt," and causes the CPU to receive an interrupt. Interrupts cause the CPU to stop executing code in user mode and switch to kernel mode, where more privileged instructions can take place.

There are many different possible interrupts, each identified by an index into a table called the "**Interrupt Descriptor Table**." Luckily, it is standard to not give each syscall its own entry, and instead access all syscalls through index 0x80, with the system call number stored in a register.

So, once the registers (arguments) are configured to your liking to perform a given syscall, and the system call number is stored in the proper register, the following instruction can be used to cause a system call on both Linux and pwnyOS:

int $0x80

Why do we need them? Why not just call OS methods directly?

For obvious security reasons, memory belonging to the kernel cannot be read, written to, or executed from userspace. We need a way to tell the CPU that we are explicitly asking the OS to do something for us, and that mechanism is the system call/ interrupt.

Syscall ABI

On pwnyOS, system calls are defined slightly differently than they are on Linux. Syscalls in pwnyOS use EAX to define the syscall number, and can take up to 3 arguments. Those 3 arguments are sent in EBX, ECX, and EDX. The return value will be sent in EAX after the syscall is finished (immediately after the "int $0x80" instruction).

System calls may change any of these 4 registers, so you should save them before calling a system call if you want to keep their current contents!

Footnote 1: Kernel Stacks and Context Switching on x86

You might be thinking, why can't we just pass syscall arguments on the stack like we do for normal functions?

On i386-style processors (32-bit x86), a special CPU data structure is configured by the kernel before any user processes run called the "Task State Segment," or TSS. The TSS does a variety of things, but most notably is it defines where the kernel stack resides. Whenever an interrupt (INT instruction, or hardware-generated interrupt) occurs, and kernel code is called, the stack is switched from the user stack to a predefined kernel stack. This is done by setting esp to whatever esp0 is set to in the TSS.

So, immediately after a system call happens, the stack is automatically swapped to a kernel stack, and the old stack pointer is pushed to the kernel stack, where it will be restored after the kernel is done with this call. Typically, processes will all have a unique kernel stack associated with them, and that region of memory is used whenever kernel functions are executing on behalf of that process.

The opposite of INT is **IRET**, or a special instruction called "interrupt return." IRET does exactly what it sounds like- it is a special form of the RET instruction designed for exiting interrupt (kernel) contexts.

When context switches happen, lots more than just the user stack is pushed to the new kernel stack- the process code and data segments, the EFLAGS register, process ESP, and process EIP values are all pushed (not in that order). These fields are then popped off the stack and used by IRET to exit the kernel context and return to user space.

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).