Modifying System Call Arguments With ptrace

As part of the ongoing effort we are doing in Canonical to snappify the world, we are trying to make available more and more software as easily-installable, secure, snaps. One of the things `snapd` does to isolate applications is to install snaps in separate folders in `/snap/<mysnap>`, and also creating `$HOME/snap/<mysnap>` for storing the snap data. Unfortunately, this changes where many applications expects data files to be as per the usual Unix conventions.

Solving this usually implies maintaining patches that change the file paths. If the `$SNAP` environment variable is set, the patches use it to find files in the right locations. However, this is cumbersome as upstreams are not always willing to accept the patches, at least until snap package format gets more popular. Also, in some cases we might be interested in snappifying proprietary software, where patches are not an option.

To solve these problems, Michael Terry created a library that uses the LD_PRELOAD trick to intercept calls to glibc. Although the library works really well, it has the disadvantage of not working in all cases, like on programs that perform syscalls directly without using the C library, or on statically compiled binaries.

In this post, I have explored an alternative to using LD_PRELOAD that would solve these issues by going to a lower level and intercepting syscalls using the `ptrace` syscall.

`ptrace` is used by programs like `gdb` or `strace` for debugging purposes. It is a swiss knife tool that lets us access a process' memory and registers (the tracee) from another process (the tracer). There are many good tutorials around that

explain how to use it, like this or this (from which I have borrowed parts of the code), so here I will focus on how it can be used to modify arbitrary syscall arguments. The concrete problem I had at hand was how to change a call to, say, `open()` so it ends up opening a file in a different path to the one originally specified by the tracee.

I have developed a proof of concept for this that can be found in github. The code is specific to x86_64, although the concepts behind are applicable to other architectures. As a word of caution, I have preferred to not do as many checks as I should to have cleaner code, so please do not consider it production-ready. I will go now through the code, function by function. Skipping the include directives and the forward declarations we get to `main()`:

```
int main(int argc, char **argv)
{
    pid_t pid;
    int status;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <prog> <arg1> ... <argN>\n", arg
        return 1;
    }

    if ((pid = fork()) == 0) {
        ptrace(PTRACE_TRACEME, 0, 0, 0);
        kill(getpid(), SIGSTOP);
        return execvp(argv[1], argv + 1);
    } else {
        waitpid(pid, &amp;status, 0);
        ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD);
        process_signals(pid);
        return 0;
    }
}
```

Here we do all the usual stuff needed when we want to trace a process from the start: we call `fork()`, then the child executes `ptrace()` with request `PTRACE_TRACEME` to indicate that it is willing to be traced. After that, it sends itself a `SIGSTOP` signal, which makes it stop (the `execve` call [1] will be performed later). At that point, the parent process, which was waiting for a signal from the child, re-starts. The first thing it does is setting `ptrace` option `PTRACE_O_TRACESYSGOOD`, which makes the kernel set bit 7 in the signal numbers so we can easily distinguish between system call traps and normal traps. After that, it calls `process_signal()`, which is defined as

```
static void process_signals(pid_t child)
{
    const char *file_to_redirect = "ONE.txt";
    const char *file_to_avoid = "TWO.txt";

    while(1) {
        char orig_file[PATH_MAX];

        /* Wait for open syscall start */
        if (wait_for_open(child) != 0) break;

        /* Find out file and re-direct if it is the target */

        read_file(child, orig_file);

        if (strcmp(file_to_avoid, orig_file) == 0)
            redirect_file(child, file_to_redirect);

        /* Wait for open syscall exit */
        if (wait_for_open(child) != 0) break;
    }
}
```

This function is the main loop of the tracer. It waits for the `open()` syscall from the child to be started, and for it to exit (`ptrace` signals us both events) by calling `wait_for_open()`. When an `open()` call is detected, we read the file that the child wants to open in `read_file()`, and then we compare with string "TWO.txt". If there is a match, we change the syscall arguments so "ONE.txt" is opened instead. Next we analyze the different functions that perform the low level stuff, and that contain architecture specific parts:

```
static int wait_for_open(pid_t child)
{
    int status;

    while (1) {
        ptrace(PTRACE_SYSCALL, child, 0, 0);
        waitpid(child, &status, 0);
        /* Is it the open syscall (sycall number 2 in x86_64)? */
        if (WIFSTOPPED(status) && WSTOPSIG(status) & 0x80 &&
            ptrace(PTRACE_PEEKUSER, child, sizeof(long)*ORIG_RAX, 0
            return 0;
        if (WIFEXITED(status))
            return 1;
    }
}
```

`wait_for_open()` is executed until an `open()` system call is detected. By calling `ptrace` with argument `PTRACE_SYSCALL`, we let the child continue until the next signal or syscall enter/exit. The first time this happens the child, which was stopped after sending itself `SIGSTOP`, continues its execution and calls `execve()`.

The parent then waits for signals from the child. If the child has stopped due to the signal, the signal number has the 7th bit set (should happen if the signal was triggered due to a syscall as we have set `PTRACE_O_TRACESYSGOOD` option), and it is the `open()` syscall (system call number 2 for x86_64), then we return with status 0. If the child has actually exited, the return value is 1. If nothing of this happens, we wait for the next signal. Here we are using `PTRACE_PEEKUSER` request, which lets us access the tracee user area. This area contains an array with the general purpose registers, and we use offsets defined in `<sys/reg.h>` to access them. When performing a syscall, `RAX` register contains the syscall number. However, we use `ORIG_RAX` offset to grab that number instead of the also existing `RAX` offset. We do this because `RAX` is also used to store the return value for the syscall, so the kernel stores the syscall number with offset `ORIG_RAX` and the return value with offset `RAX`. This needs to be taking into account especially when processing the exit of the system call. More information can be found here.

Next function is

```
static void read_file(pid_t child, char *file)
{
    char *child_addr;
    int i;

    child_addr = (char *) ptrace(PTRACE_PEEKUSER, child, sizeof(lon

    do {
        long val;
        char *p;

        val = ptrace(PTRACE_PEEKTEXT, child, child_addr, NULL);
        if (val == -1) {
            fprintf(stderr, "PTRACE_PEEKTEXT error: %s", strerror(e
            exit(1);
        }
        child_addr += sizeof (long);

        p = (char *) &val;
        for (i = 0; i < sizeof (long); ++i, ++file) {
            *file = *p++;
            if (*file == '\0') break;
        }
    } while (i == sizeof (long));
}
```

The `read_file()` function uses `PTRACE_PEEKUSER` as in the previous function to retrieve the first argument to the call function, which is the address of the string with the file name. This parameter is stored in the `RDI` register. Then it uses

`PTRACE_PEEKTEXT` `ptrace` request, which lets us copy over data from the traced process' memory. This is performed by words, so we do this in a loop until we find a null byte that indicates the end of the string.

The last function is

```
static void redirect_file(pid_t child, const char *file)
{
    char *stack_addr, *file_addr;

    stack_addr = (char *) ptrace(PTRACE_PEEKUSER, child, sizeof(lon
    /* Move further of red zone and make sure we have space for the
    stack_addr -= 128 + PATH_MAX;
    file_addr = stack_addr;

    /* Write new file in lower part of the stack */
    do {
        int i;
        char val[sizeof (long)];

        for (i = 0; i < sizeof (long); ++i, ++file) {
            val[i] = *file;
            if (*file == '\0') break;
        }

        ptrace(PTRACE_POKETEXT, child, stack_addr, *(long *) val);
        stack_addr += sizeof (long);
    } while (*file);

    /* Change argument to open */
    ptrace(PTRACE_POKEUSER, child, sizeof(long)*RDI, file_addr);
}
```

The `redirect_file()` function is the most interesting one of this program. It modifies the argument to the open system call, forcing the child to open a different file to the one it originally specified. The main problem with this is that we need to modify the child's memory space so the new file name is used by the kernel. We can change the tracee's memory easily by using `PTRACE_POKETEXT`, the issue is, where can we store it?

A possible option is to save the original string using `PTRACE_PEEKTEXT`, then overwrite it by using `PTRACE_POKETEXT`. When we get called after the sycall exits, we copy back the original data. This can work fine in some cases, but it can be problematic if the new file name is longer than the original. We could be overwriting data that is used by other threads, which are not necessarily stopped so they could access that data while the kernel is processing the call. Or that data we are overwriting could be part of another parameter to the syscall, which would

not happen for `open()`, but it is possible for other syscalls like `link()`. Finally, there is also the possibility that the string we are trying to modify is in a read only segment. Therefore, this is not a safe option.

After noticing this, I considered the option of adding a read-write segment to the binary under study, or to resize an existing one. However, I found there are not that many tools to do this, and those that apparently could do the job like ERESI, were not very intuitive [2]. Also, we would need to find out where the new segment gets loaded to know where to write, which would complicate the code. Furthermore, I wanted to avoid modifying the binary if possible.

Finally, I concluded that the stack was exactly what I needed: it is of course RW, and a reference address can be found by simply looking at the `RSP` register. What we have to do is make sure we write in a safe part of the stack. This can be performed by writing to addresses lower than the `RSP` (that is, the free part of the stack). To achieve this, we "reserve" stack memory so we can write a string of up to `PATH_MAX` length, and we add up 128 bytes for the red zone (this size is specified by the x86_64 ABI). Note also that syscalls do not write on the process stack: one of the very first things that is done by the Linux kernel syscalls entry point is to switch `RSP` to a kernel stack. This approach has also the advantage of being automatically thread-friendly, as each one has its own stack. On the other hand, there is the possibility of writing outside the stack. However that risk is quite small nowadays, as stacks of user space programs tend to be big and typically auto-grow on page fault. Another advantage is that we do not need to save and recover memory areas at enter/exit of the syscall, as the tracee should not write anything in the used memory area.

Once it is decided where to write, the implementation is straightforward: first we use `PTRACE_PEEKUSER` to get the `RSP` value of the tracee. Then, we write the new file name to a pointer lower than the `RSP` calculated as explained in the previous paragraph. The data is written by using `PTRACE_POKETEXT`, word by word. Finally, we change the child's `RDI` register so it points to the new address.

Now we can give the program a try. I created a couple of files with content:

```
$ cat ONE.txt
This is ONE.txt
$ cat TWO.txt
```

```
This is TWO.txt
```

Executing the same `cat` command using `redirect` we have:

```
$ gcc redirect.c -o redirect
$ ./redirect cat ONE.txt
This is ONE.txt
$ ./redirect cat TWO.txt
This is ONE.txt
```

Things work as publicized: we modify the file opened by the `cat` in case it tries to show the content of "TWO.txt".

Conclusions

As has been seen, the code to make this is remarkably small, which shows the power of the `ptrace` call. There are indeed parts of this that are very architecture specific, but that is mostly the name of the registers and maybe the red zone size, so it should be relatively straightforward to make it multi-architecture by adding some macros.

Another appreciation is that the example is for the `open()` syscall, but this technique can be applied to arbitrary arguments which are passed to any syscall as pointers to data in the traced process.

To finish, the main drawback for this solution is performance, as we have to stop (twice) for each syscall invoked by the child, with all the context switches that implies. A possible solution would be to use `ptrace` in combination with seccomp and seccomp Berkeley Packet Filters, which apparently make possible for the tracer to specify the syscalls that would provoke a trap. That would be, however, matter for another post.

1. The C standard guarantees that the `argv` array ends with a `NULL` pointer, so we can simply use `argv + 1` as second argument to `execve`. ↵
2. Note that to modify/add sections is actually easy using `objcopy`, but here we need to change the elf binary segments to modify what gets loaded into memory. ↵