Ret to Argc

What happens when you return from a bare program?

2026-04-09

> Suppose we are working on a `x86_64-unknown-linux-gnu` machine.

Crash Demo

What happens if a Linux process returns from `_start`?

```c
// ret.c
int _start() {
  return 0;
}
```

```
$ gcc -nostdlib -static -o ret ret.c
$ ./ret
Segmentation fault (core dumped)
$ echo $?
139
```

It crashes because `ret` does not return to a caller here; it pops the process's initial `argc` value and treats it as a code address.

What happens in a Normal C Program

In a normal C program, `_start` does not return: it hands control to `__libc_start_main`, which in turn calls `main` and then exits via syscall `exit`.

At the source level, we talk about `main` as the entry point. At the binary level, that role belongs to `_start`. `_start` calls `__libc_start_main`, which in turn calls `main`.

When `main` returns, its return value is passed back to `__libc_start_main`, which then passes it to the kernel via the `exit` system call. In normal execution, `_start` (and `__libc_start_main`) never returns.

What `ret` Sees at `_start`

> RET pops `[RSP]` into `RIP`, and at process entry `[RSP]` holds `argc`.

When the kernel enters `_start` on Linux x86_64, the top of the stack is not a saved return address; it is `argc`.

At that point, the stack looks like this:

```
High    | AUX, Strings, ...      |
        | NULL                   |
  ^     | envp[n]...envp[0]      |
  |     | NULL                   |
        | argv[n]...argv[0]      |
[RSP]   | argc                   |
Low     | (Undefined)            |
```

A normal `ret` instruction performs a very small sequence of actions:

1. the kernel jumps to `_start`
2. `_start` executes `ret`
3. the CPU loads `RIP <- [RSP]`
4. here `[RSP]` is `argc`
5. in the demo, `argc = 1`, so the CPU tries to jump to address `0x1`

That last jump faults immediately, which is why the bare program crashes with `SIGSEGV`.

Can `argc` Be a Valid Return Address?

If we can pass enough arguments to `_start` so that the value in the `argc` slot is a valid executable address, we can actually make it return.

The two main questions are:
1. How many arguments can we pass to `_start`?
2. What is the lowest valid executable address we have in the process?

How Many Arguments Can We Pass?

On a typical Linux system:

- argument is capped to 1/4 of the process stack limit
- the default stack limit is often 8 MiB (which is configurable via `ulimit -s`)
- each argument costs at least 9 bytes of stack space (a pointer plus a null terminator)

So we can pass about `230,000` arguments.

What is the lowest valid executable address?

A normal non-PIE ELF binary has a default base address of `0x400000` (`4,194,304`), which is far higher than our `230,000` limit. Because of this, we can't directly jump to the code section of a binary.

Linux exposes a sysctl called `vm.mmap_min_addr`, which sets the minimum virtual address a process may map. On typical systems this is `65536`, or `0x10000`, which is well within our reach. We can map a executable page at `0x10000` and put our code there.

> The `vm.mmap_min_addr` sysctl is a security feature that prevents mapping memory at very low addresses, which can help mitigate certain types of exploits. However, it

> also means that we cannot map memory at address `0x0`. Fortunately, `0x10000` is still a valid executable address that we can use for our purposes.

Building the Demo

This program maps one executable page at `0x10000`, writes `exit(42)` there, and then executes `ret`.

bare_ret.s

```
; bare_ret.s
.intel_syntax noprefix
.global _start

.equ TARGET, 0x10000

.section .text
_start:
    mov rax, 9        ; sys_mmap
    mov rdi, TARGET
    mov rsi, 4096
    mov rdx, 7        ; PROT_READ | PROT_WRITE | PROT_EXEC
    mov r10, 0x32     ; MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED
    mov r8, -1
    xor r9d, r9d
    syscall

    cmp rax, -4095
    jae .failed

    ; b8 3c 00 00 00   mov eax, 60
    ; bf 2a 00 00 00   mov edi, 42
    ; 0f 05            syscall
    movabs rbx, 0x002abf0000003cb8
    mov qword ptr [TARGET + 0], rbx
    mov dword ptr [TARGET + 8], 0x050f0000

    ret

.failed:
    mov eax, 60
    mov edi, 1
    syscall
```

To invoke the program with that many arguments, it is convenient to write a small launcher.

launcher.c

```c
// launcher.c
#define _GNU_SOURCE
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define TARGET_ARGC 0x10000

int main(void) {
    const char *prog = "./bare_ret";
    const size_t argc_target = TARGET_ARGC;

    char **argv = calloc(argc_target + 1, sizeof(char *));
    if (!argv) {
        perror("calloc argv");
        return 1;
    }

    argv[0] = (char *)prog;
    for (size_t i = 1; i < argc_target; i++) {
        argv[i] = (char *)"";
    }
    argv[argc_target] = NULL;

    char *envp[] = { NULL };
    execve(prog, argv, envp);

    fprintf(stderr, "execve failed: %s\n", strerror(errno));
    free(argv);
    return 1;
}
```

```
$ gcc -nostdlib -static -o bare_ret bare_ret.s
$ gcc -o launcher launcher.c
$ ./launcher && echo $?
42
```

What Actually Happens

The program didn't crash. Instead, it successfully returned from `_start` and exited with status `42`. The exact execution flow is:

1. `launcher` allocates an `argv` array with `0x10000` entries, counting the program name
2. `execve` starts `bare_ret` with `argc = 0x10000`
3. the kernel places that `argc` value at `[RSP]` before entering `_start`
4. `_start` calls `mmap` to create an executable page at `0x10000`
5. `_start` writes machine code for `exit(42)` into that page

6. `_start` executes `ret`
7. the CPU pops `0x10000` from `[RSP]` into `RIP`
8. execution continues at `0x10000`, which performs the `exit(42)` syscall

And that is the end of the story.

Edited on 2026-04-09