

# 上海電力大學

SHANGHAI UNIVERSITY OF ELECTRIC POWER



## *FAULT TOLERANCE*

題目： *Volt OS 錯誤注入实验报告书*  
(*ERROR INJECTION Method*)

*Under the on-campus supervision of*

*Dr. Man Xu, Dr. Zhen Wang*

*Under the off-campus supervision of*

*Dr. Wei Zhang*

*Tested by Lei Tang, Software Engineering*

# 头

在软件工程领域，软件的可靠性和稳定性是衡量其质量的关键指标。正如著名软件工程专家 Watts S. Humphrey 所言：“软件的质量是构建出来的，而不是测试出来的，但严格的测试却是确保高质量软件必不可少的环节。”为了确保软件在实际运行中能够承受各种异常情况，故障注入测试成为了一项至关重要的技术手段。

软件故障注入已测试系统可靠性的技术，一般分为基于代码变异的缺陷注入和缺陷表现（错误）注入。诚如软件工程权威学者 Barry Boehm 所说：“对软件的深入理解和把控需要从代码的根源以及实际运行表现双管齐下。”基于代码变异的缺陷注入正是通过对软件代码进行特定的变异操作，模拟可能出现的代码缺陷，从根源上探寻软件潜在问题；而缺陷表现（错误）注入则是直接在软件运行过程中注入错误表现，以观察软件的应对能力，在实际运行层面来检验软件的健壮性。

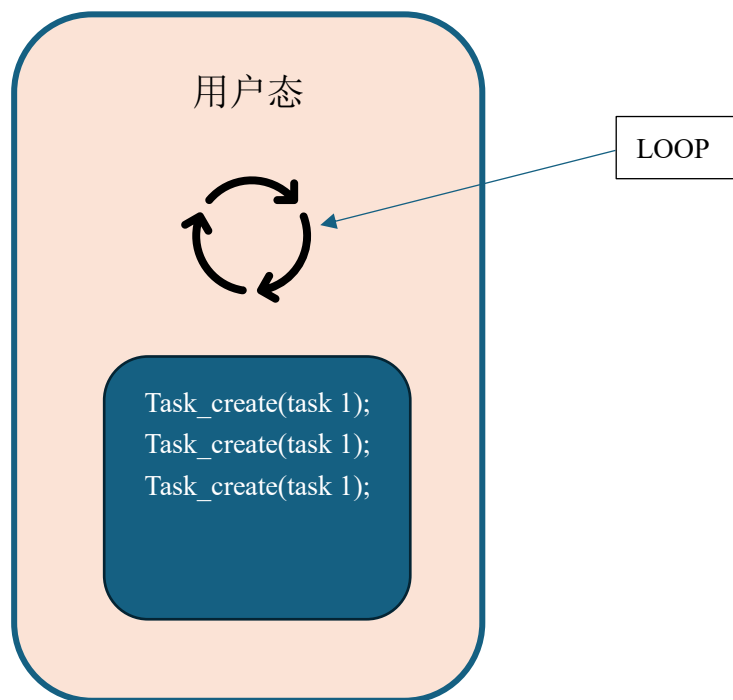
在实际的软件开发过程中，软件编写人员总是不可避免地会写出与预期不符、不理想的代码。软件工程大师 Fred Brooks 曾强调：“人月神话告诉我们，软件开发中的复杂性和不可预测性无处不在，即使最优秀的程序员也难以完全避免失误。”这些潜在的问题代码可能在软件投入使用后引发各种故障，影响软件的正常运行。因此，为了确保代码质量，我们不得不对一个编码完成的系统进行故障注入测试。通过这种方式，可以提前发现软件中可能存在的问题，并及时进行修复，从而提高软件的可靠性和稳定性，为用户提供更加可靠的软件产品。

## 一、修改函数入口环境

这是 sched.c 文件的 task\_create 模块，用于创建任务。

```
int task_create(void (*start_routine)(void))
{
    if (_top < MAX_TASKS) {
        ctx_tasks[_top].sp = (reg_t) &task_stack[_top][STACK_SIZE];
        ctx_tasks[_top].pc = (reg_t) start_routine;
        _top++;
        return 0;
    } else {
        return -1;
    }
}
```

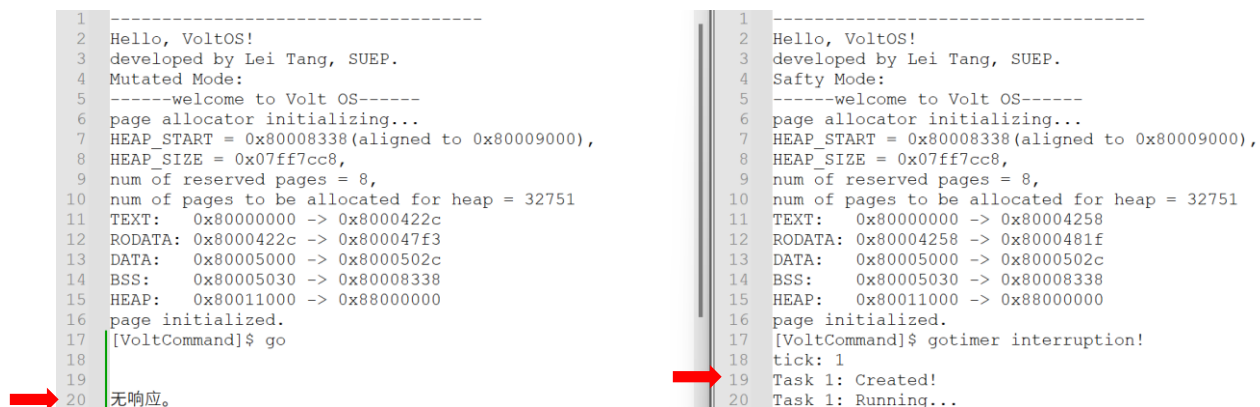
在这里，我们在 user.c(用于创建用户态进程的 c 文件)的调用 task\_create() 模块人为地引入错误模块如下：



```
void os_main(void)
{
    while (1)
    {
        // 空转
    }

    task_create(user_task0);
}
```

我们可以通过实验来观察此时 kernel 的运行情况。如下图 1 所示：



```
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Mutated Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x8000422c
12 RODATA: 0x8000422c -> 0x800047f3
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ go
18
19
20 无响应.
```

```
1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Saftey Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004258
12 RODATA: 0x80004258 -> 0x8000481f
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ gotimer interruption!
18 tick: 1
19 Task 1: Created!
20 Task 1: Running...
```

图 1 在程序入口处引入故障模块运行报告对比

## 二、修改函数运行时传入的参数

在函数运行的时候，往往是以某种功能模块执行某项任务的。一般情况而言，如果某个带参列表的函数肯定希望的是传入的参数是正确的，这样来的函数体执行后才是有意义的、安全的。

但是，现实中的程序往往并没有那么如人所愿。程序编写者往往会由于人工失误的原因引发代码缺陷。其中，相对常见的就是参数缺陷问题。

以下面的 sched.c 文件的模块为例：

我们可以看到，对于 sched.c 文件的核心接口是 schedule 函数。

```
void schedule()
{
    if (_top <= 0) {
        panic("Num of task should be greater than zero!");
        return;
    }

    _current = (_current + 1) % _top;
    struct context *next = &(ctx_tasks[_current]);
    switch_to(next);
}
```

通过对于任务队列的 top 和 current 的调整，以动态地调整调度的任务。而对于 schedule 模块，核心模块是 switch\_to 模块，其希望的参数是一个上下文类型的指针。对于该参数而言，如果传入的指针被污染，那么必然会使得任务切换使能部分失效。

通过错误注入后的代码如下：

```

void schedule()
{
    if (_top <= 0) {
        panic("Num of task should be greater than zero!");
        return;
    }

    _current = (_current + 1) % _top;
    struct context *next = &(ctx_tasks[_current]);
    switch_to(NULL);
}

```

可以观察到，我们将 switch\_to 的函数参数修改为了 NULL。运行过程如下图 2：

```

1 -----welcome to Volt OS-----
2 page allocator initializing...
3 HEAP_START = 0x80008338(aligned to 0x80009000),
4 HEAP_SIZE = 0x07ff7cc8,
5 num of reserved pages = 8,
6 num of pages to be allocated for heap = 32751
7 TEXT: 0x80000000 -> 0x80004258
8 RODATA: 0x80004258 -> 0x8000481f
9 DATA: 0x80005000 -> 0x8000502c
10 BSS: 0x80005030 -> 0x80008338
11 HEAP: 0x80011000 -> 0x88000000
12 page initialized.
13 [VoltCommand]$ go
14 系统崩溃了。
15
16
17

```

```

1 -----welcome to Volt OS-----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Safty Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004258
12 RODATA: 0x80004258 -> 0x8000481f
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ gotimer interruption!

```

图 2 修改函数参数列表的值后的运行报告对比

### 三、修改执行函数体后的返回值

函数执行时，其执行的函数体是对于输入数据的处理算法的直接体现。若我们假设：

1. 编程人员对于算法的编码是正确的。
2. 函数没有被注入脏参数

函数可以分为有返回值的函数、无返回值的函数。对于有返回值的函数我们可以通过接收其返回值，然后继续执行其他的代码逻辑。但是，对于无返回值的函数，一般我们是对于某个地址作出修改动作，或是完成初始化等工作。当我们对无返回值的参数增加打印函数，便可以监控函数体的执行过程。

在这里，我们要做出修改函数体的返回值，以让其他模块调用时出错。利用其自身耦合性和相互调用的接口特性进行故障注入：

在注入前：

```

int task_create(void (*start_routine)(void))
{
    if (_top < MAX_TASKS) {
        ctx_tasks[_top].sp = (reg_t) &task_stack[_top][STACK_SIZE];
        ctx_tasks[_top].pc = (reg_t) start_routine;
        _top++;
        return 0;
    }
}

```

```

    } else {
        return -1;
    }
}

```

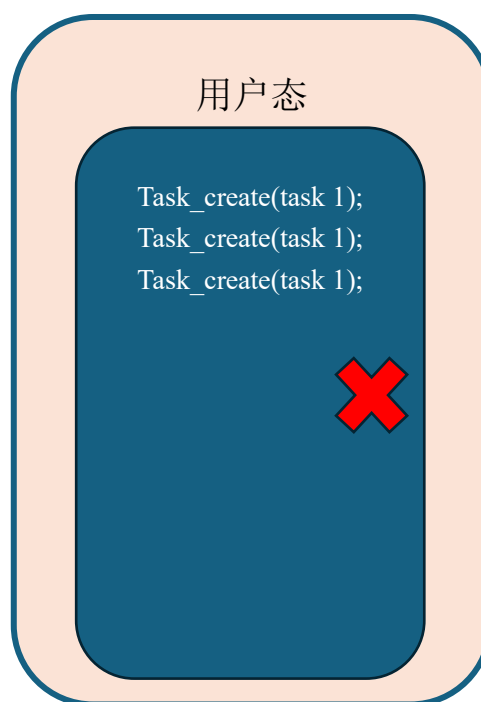
在注入后:

```

int task_create(void (*start_routine)(void))
{
    if (_top < MAX_TASKS) {
        ctx_tasks[_top].sp = (reg_t) &task_stack[_top][STACK_SIZE];
        ctx_tasks[_top].pc = (reg_t) start_routine;
        _top++;
        return NULL;
    } else {
        return NULL;
    }
}

```

对于 task\_create() 模块，我们在后续的用户态创建任务进程的时候需要调用之。  
其调用图如下：



其运行结果如下图 3 所示：

```

1 -----welcome to Volt OS-----
2 page allocator initializing...
3 HEAP_START = 0x80008338(aligned to 0x80009000),
4 HEAP_SIZE = 0x07ff7cc8,
5 num of reserved pages = 8,
6 num of pages to be allocated for heap = 32751
7 TEXT: 0x80000000 -> 0x80004258
8 RODATA: 0x80004258 -> 0x8000481f
9 DATA: 0x80005000 -> 0x8000502c
10 BSS: 0x80005030 -> 0x80008338
11 HEAP: 0x80011000 -> 0x88000000
12 page initialized.
13 [VoltCommand]$ go
14
15 系统崩溃了。
16
17
18
19
20
21

```

```

1 -----
2 Hello, VoltOS!
3 developed by Lei Tang, SUEP.
4 Safty Mode:
5 -----welcome to Volt OS-----
6 page allocator initializing...
7 HEAP_START = 0x80008338(aligned to 0x80009000),
8 HEAP_SIZE = 0x07ff7cc8,
9 num of reserved pages = 8,
10 num of pages to be allocated for heap = 32751
11 TEXT: 0x80000000 -> 0x80004258
12 RODATA: 0x80004258 -> 0x8000481f
13 DATA: 0x80005000 -> 0x8000502c
14 BSS: 0x80005030 -> 0x80008338
15 HEAP: 0x80011000 -> 0x88000000
16 page initialized.
17 [VoltCommand]$ gotimer interruption!
18 tick: 1
19 Task 1: Created!
20 Task 1: Running...
21 Task 1: Running...

```

图 3 修改返回值注入运行对比