# SYSU-160 CrashCrawler阶段性报告

## 成员信息

**学校：中山大学**

**学院：计算机学院**

**专业：计算机科学与技术**

| 姓名 | 年级 | 邮箱 |
| --- | --- | --- |
| 付恒宇 | 大二 | 2941845883@qq.com |
| 洪瑞鹏 | 大二 | 2634367909@qq.com |
| 唐喆 | 大二 | tangzh33@mail2.sysu.edu.cn |


## 项目调研

### 项目需求分析

项目要求实现的崩溃收集组件，要求我们正确的拦截到崩溃的进程。

我们初步考虑了Linux提供的core dump机制。

### coredump机制的分析

core dump机制是一种静态进程状态收集方法。

**优势**

在core文件限制大小允许情况下，可以把异常终止的进程的内存镜像写到core文件里面，可以借助调试工具（gdb，lldb）定位问题所在具体代码，为程序debug提供很大帮助。

**劣势**

只能在开发调试的时候启用core dump机制，没办法方便的监控操作系统全局的异常进程。首先是core文件限制大小默认为0，我们只能通过ulimit手动地设置core文件大小（只对当前shell和该shell启动的进程有效），终端重启了或者另一个用户启动了另一个进程，还是默认无法产生core文件，没法实时且全局地追踪进程异常退出。

得到的结论是core dump机制并不适合实时地拦截并获取崩溃进程的相关信息。

### eBPF和LKM的对比分析

所以我们转而采用Linux提供的动态的内核信息获取工具。Linux为我们提供了tracepoint，kprobe，uprobe等机制，这些机制为我们提供了一个事件触发式的获取某个内核函数或者是用户级函数发生时的上下文的功能。

而由于用户态我们通过上述方法可以获取的信息时较为固定的，比较有限，可能无法收集到足够有用的信息。所以我们可以尝试进入到内核态去获取信息。我们可以通过KVM或者是eBPF来进入内核态获取信息。

图表引用自这里

| 维度 | Linux 内核模块 | eBPF |
| --- | --- | --- |
| kprobes/tracepoints | 支持 | 支持 |
| **安全性** | 可能引入安全漏洞或导致内核 Panic | 通过验证器进行检查，可以保障内核安全 |
| 内核函数 | 可以调用内核函数 | 只能通过 BPF Helper 函数调用 |
| 编译性 | 需要编译内核 | 不需要编译内核，引入头文件即可 |
| 运行 | 基于相同内核运行 | 基于稳定 ABI 的 BPF 程序可以编译一次，各处运行 |
| 与应用程序交互 | 打印日志或文件 | 通过 perf_event 或 map 结构 |
| 数据结构丰富性 | 一般 | 丰富 |
| **入门门槛** | 高 | 低 |
| **升级** | 需要卸载和加载，可能导致处理流程中断 | 原子替换升级，不会造成处理流程中断 |
| 内核内置 | 视情况而定 | 内核内置支持 |

由上表我们和指导老师讨论过后得出结论：采用eBPF来拦截和获取崩溃进程的信息。

## eBPF存在的限制

eBPF为了提供安全性而牺牲了很多灵活性，比如BPF代码逻辑比较简单（较低内核版本存在指令数限制，堆栈大小存在限制等）。

但是经过讨论，在这些限制下，我们仍旧可以完成进程异常退出时的信息收集并返回给用户态。

## 可行性分析

在挂载了debugfs的前提下

1. sched_process_exit是eBPF程序可以挂载的点，通过libbpf提供的两个helper函数，bpf_get_current_task和bpf_get_stackid可以得到当前进程的backtrace

   并且由下图（来自linux源码task_struct里面的部分代码），可以得到trace_sched_process_exit(tsk)的挂载点是在进程退出的主要工作都完成了的，并且现场还没被销毁的时刻。

```
971         /* sync mm's RSS info before statistics gathering */
972         if (tsk->mm)
973                 sync_mm_rss(tsk->mm);
974         group_dead = atomic_dec_and_test(&tsk->signal->live);
975         if (group_dead) {
976                 hrtimer_cancel(&tsk->signal->real_timer);
977                 exit_itimers(tsk->signal);
978                 if (tsk->mm)
979                         setmax_mm_hiwater_rss(&tsk->signal->maxrss, tsk->mm);
980         }
981         acct_collect(code, group_dead);
982         if (group_dead)
983                 tty_audit_exit();
984         audit_free(tsk);
985
986         tsk->exit_code = code;
987         taskstats_exit(tsk, group_dead);
988
989         exit_mm(tsk);
990
991         if (group_dead)
992                 acct_process();
993         trace_sched_process_exit(tsk);
994
995         exit_sem(tsk);
996         exit_shm(tsk);
997         exit_files(tsk);
998         exit_fs(tsk);
999         check_stack_usage();
1000        exit_thread();
1001
1002        /*
```

2. linux man signal(7) 提供了进程退出时候的信号类型代表的可能存在问题和内核处理进程方法。
3. exitcode 分析 https://www.linuxdoc.org/LDP/abs/html/exitcodes.html
4. 通过ebpf可以得到stacktrace

***int bpf_get_stack(struct pt_regs *regs, void *buf, u32 size, u64 flags)***

***Description***

Return a user or a kernel stack in bpf program provided buffer. To achieve this, the helper needs *ctx*, which is a pointer to the context on which the tracing program is executed. To store the stacktrace, the bpf program provides *buf* with a nonnegative *size*.

The last argument, *flags*, holds the number of stack frames to skip (from 0 to 255), masked with **BPF_F_SKIP_FIELD_MASK**. The next bits can be used to set the following flags:

***BPF_F_USER_STACK***

Collect a user space stack instead of a kernel stack.

***BPF_F_USER_BUILD_ID***

Collect buildid+offset instead of ips for user stack, only valid if **BPF_F_USER_STACK** is also specified.

**bpf_get_stack**() can collect up to **PERF_MAX_STACK_DEPTH** both kernel and user frames, subject to sufficient large buffer size. Note that this limit can be controlled with the **sysctl** program, and that it should be manually increased in order to profile long user stacks (such as stacks for Java programs). To do so, use:

```
# sysctl kernel.perf_event_max_stack=<new value>
```

***Return***

a non-negative value equal to or less than size on success, or a negative error in case of failure.

# 项目设计思路

1. 在崩溃收集组件加载阶段就可以收集宿主机的静态信息（硬件架构，操作系统版本）
2. 通过eBPF拦截到异常退出的进程
3. 通过eBPF程序可以从内核态获取到该进程的动态信息（exitcode，退出信号，还有stacktrace等）
4. 收集该进程相关软件包版本和依赖（该部分实现思路不清晰，较难获取）
5. 最后根据已收集信息整合并分析，进而生成崩溃报告。

未来的进一步实现可能主要着重于更多可能有用的信息收集，以及生成更具指导意义的崩溃日志。

# 项目实现和开发状态

截止到2022年5月14日，我们完成了通过eBPF程序实现了崩溃进程异常退出时的内核态信息收集。

通过Libbpf + CO:RE 来编写了一个eBPF程序，收集了进程异常退出（exitcode=0的除外）的exitcode和退出信号，需要进一步完善的是通过maps或者perf_event来把收集到的信息（包括后面收集的函数调用栈信息）回传给用户态程序（数据分析部分）。

如图所示，我们可以跟踪到任意进程非正常退出的exitcode和signal，在后面可以用于数据分析和崩溃报告生成。

```
^Chrpccs[17:28]:~/test$ sudo ./hello
[sudo] password for hrpccs:
Successfully started! Tracing /sys/kernel/debug/tracing/trace_pipe...
            cat-15385   [000] d... 111234.288487: bpf_trace_printk: comm:
cat exitcode:0 signal:2

             sh-15384   [003] d... 111234.296551: bpf_trace_printk: comm:
sh exitcode:0 signal:2

         acrash-19445   [006] d... 111255.398718: bpf_trace_printk: comm:
acrash exitcode:0 signal:2

         acrash-19871   [006] d... 111324.987866: bpf_trace_printk: comm:
acrash exitcode:0 signal:2

            man-19980   [003] d... 111354.596262: bpf_trace_printk: comm:
man exitcode:0 signal:13

         acrash-20119   [003] d... 111379.329249: bpf_trace_printk: comm:
acrash exitcode:0 signal:2
```

也就是说我们目前大致完成了崩溃进程退出的拦截还有信息的获取。

后面的数据分析和崩溃日志的生成还需要时间完成。

# 遇到的问题和解决方法

1. 在项目需求分析和设计阶段，我们遇到的主要问题是关于进程软件包版本和依赖的获取问题。（未解决）
    - 对于动态链接可执行文件，我们可以lsof来分析该进程的内存布局，从中可以获取其动态库的依赖和相应的版本信息。
    - 但是对于其它的一些软件依赖信息，存在获取困难。对于一些通过源码编译自定义安装的软件，很难在进程异常退出的时候获取到软件包版本依赖的信息。

    目前还没有办法解决，希望项目导师可以给出一点指导意见。

    并且对于软件包版本和软件包依赖信息对于进程崩溃的联系以及分析时候起到的作用并不是很明确。

2. 对于数据分析生成崩溃报告的阶段，我们该用怎样的策略来实现高效有指导意义的崩溃分析。（未解决）

    一种比较容易想到的方法是通过if,else语句来依据获取到的信息（exitcode，signal等）代表的比较固定的信息建立一个数据分析的库。但是这种实现显然比较低效。

# 参考资料

**eBPF学习**

[1] eBPF概念学习https://www.ferrisellis.com/content/ebpf_past_present_future/

[2] perf_event学习 https://www.cnblogs.com/pwl999/p/15535028.html

[3] Linux man bpf（2）

[4] Libbpf + CO:RE编程学习 https://github.com/DavadDi/bpf_study示例 和 https://facebookmicrosites.github.io/bpf/blog/2020/02/20/bcc-to-libbpf-howto-guide.html#field-accesses

[5] libbpf 编程示例参考 https://github.com/iovisor/bcc/tree/master/libbpf-tools

**数据分析依据**

[1] Linux man signals (7)

[2] exitcode 意义分析 https://www.linuxdoc.org/LDP/abs/html/exitcodes.html

**日志生成策略参考**

暂无