Coverage for eBPF programs

Leonardo Di Donato - 05 Feb 2022 @ FOSDEM 22 - LLVM devroom
Leonardo Di Donato
Open Source Software Engineer
Falco Maintainer
Senior eBPF Engineer @ Elastic Security

@leodido 🐝
why?

- Lot of **eBPF** for tracing and security applications out there
- Lot of **developers** approaching eBPF
- No simple way for them to get **coverage** for their **eBPF code running in the Linux kernel**
- **Test** eBPF programs via `BPF_PROG_TEST_RUN`, but not all program types are supported
- Which path my eBPF code took while running in the kernel? Which **code regions** or **branches** got evaluated and to what?
- General **lack of tooling** in the **eBPF ecosystem**
Gather **source-based code coverage** for our **eBPF applications**.

**eBPF is:**

- usually written in C
- compiled via Clang to BPF ELF `.o` files
  - LLVM BPF target
- loaded through the `bpf()` syscall
- executed by the eBPF Virtual Machine in the Linux kernel
What's source-based coverage?

- Line-level granularity is not enough
- AST → regions, branches, ...
- Better to find grasps in the code

Why is branch Coverage Important?

- There are two conditions on line 10 that form a decision: $(x > 0)$, $(y > 0)$
- Line 11 shows that "return true" was executed once
  - What was the execution path through the control flow that facilitated this?
  - What was the execution path through the control flow around this?
  - If we don’t know, we can’t be sure we are executing all paths!
- Branch Coverage tells us this!
  - How many times is each condition taken (True) or not taken (False)?

Counter Region Mapping and Instrumentation

- Counters are inserted into basic blocks of generated code mapped to source
- Counter1 instrumented to track
  - Region: 9:24 → 10:23
  - Function (line 9 → foo(1))
  - Statement: if-stmt
- Counter2 instrumented to track
  - Region: 10:18 → 10:25
  - Statement: $(y > 0)$
- Counter3 instrumented to track
  - Region: 11:0 → 11:12
  - Line coverage (line 11)
- (Counter1 – Counter3) tracks
  - Region: 12:0 → 14:0
  - Line coverage (line 13)
Source-based code coverage\(^1\) for C programs

```c
#include <stdio.h>
#include <stdint.h>

void ciao()
{
    printf("ciao\n");
}

void foo()
{
    printf("foo\n");
}

int main(int argc, char **argv)
{
    if (argc > 1)
    {
        foo();
        for (int i = 0; i < 22; i++) {
            ciao();
        }
    }
    printf("main\n");
}
```

\(^1\)for more details visit the [LLVM docs](https://llvm.org/docs/)

```
$ clang \
    -fprofile-instr-generate \
    -fcoverage-mapping \
    hello.c \
    -o hello

$ ./hello yay

$ llvm-profdata merge \
    -sparse default.profraw \
    -o hello.profdata

$ llvm-cov show \
    --show-line-counts-or-regions \
    --show-branches=count \
    --show-regions \
    -instr-profile=hello.profdata \
    hello
```
Source-based coverage

— **Efficient** and **accurate**
— Works with the existing LLVM coverage tools
— Highlights **exact regions** of code (**line:col** to **line:col**) that were skipped or executed
— Counts how many times a condition (**branches**) was taken or not (see lines 16 and 23)
— Tells us what was the **execution path** through the code
Instruments the program functions to collect execution counts
define dso_local void @foo() #0 !dbg !15 {
  %1 = load i64, i64* getelementptr inbounds ([1 x i64], [1 x i64]* @func, i64 0, i64 0), align 8, !dbg !16
  %2 = add i64 %1, 1 !dbg !16
  store i64 %2, i64* getelementptr inbounds ([1 x i64], [1 x i64]* @func, i64 0, i64 0), align 8, !dbg !16
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @str), i64 0, i64 0), !dbg !17
  ret void, !dbg !18
}

define dso_local i32 @main(i32 %0, i8** %1) #0 !dbg !19 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  %6 = alloca i32, align 4
  store i32 0, i32* %3, align 4
  store i32 0, i32* %4, align 4
  call void @llvm.dbg.declare(metadata i32* %4, metadata !26, metadata !DIExpression()), !dbg !27
  store i8** %1, i8** %5, align 8
  call void @llvm.dbg.declare(metadata i8*** %5, metadata !28, metadata !DIExpression()), !dbg !29
  %7 = load i64, i64* getelementptr inbounds ([3 x i64], [3 x i64]* @func, i64 0, i64 0), align 8, !dbg !30
  %8 = load i64, i64* getelementptr inbounds ([3 x i64], [3 x i64]* @func, i64 0, i64 0), align 8, !dbg !30
  %9 = load i32, i32* %4, align 4, !dbg !31
  %10 = icmp sgt i32 %9, 1, !dbg !33
  br %10, label %11, label %24, !dbg !34

  ; preds = %2

  %12 = load i64, i64* getelementptr inbounds ([3 x i64], [3 x i64]* @func, i64 0, i64 1), align 8, !dbg !34
  %13 = load i64, i64* getelementptr inbounds ([3 x i64], [3 x i64]* @func, i64 0, i64 1), align 8, !dbg !34
  call void @foo(), !dbg !35
  call void @llvm.dbg.declare(metadata i32* %6, metadata !37, metadata !DIExpression()), !dbg !39
  store i32 0, i32* %6, align 4, !dbg !39
  br label %14, !dbg !40
}

...
Generate coverage mappings

@__covel__EB77D490E4C6665E = linkonce_odr hidden constant <[ i64, i32, i64, i64, [9 x i8]]> <@
  i64 -1479480177954886802,
  i32 9,
  i64 0,
  i64 -662421541936037574,
  [9 x i8] "\01\00\00\00\01\00\01\02\02"
}@, section ""llvm_covfun", comdat, align 8

@_covel__5CF8C24CD818BDACu = linkonce_odr hidden constant <[ i64, i32, i64, i64, [9 x i8]]> <@
  i64 6699318801052747564,
  i32 9,
  i64 0,
  i64 -662421541936037574,
  [9 x i8] "\01\00\00\00\01\00\01\02\02"
}@, section ""llvm_covfun", comdat, align 8

@_covel__DB956436E78DD5FAu = linkonce_odr hidden constant <[ i64, i32, i64, i64, [70 x i8]]> <@
  i64 -252408102089762054,
  i32 70, i64 717562688593,
  i64 -662421541936037574,
  [70 x i8] "\01\00\02\01\05\01\09\0A\01\0F\01\09\02\01\01\09\00\01\05\01\0F\09\09\0A"
}@, section ""llvm_covfun", comdat, align 8

@__llvm_coverage_mapping = private constant { { i32, i32, i32, i32 }, [72 x i8] } <@
  { i32 0, i32 72, i32 0, [i32 4] },
  [72 x i8] "\01\0E\00D/home/leodido/workspace/github.com/leodido____/mycovexample/hello.c"
}@, section ""llvm_covmap", align 8
Demystifying the profraw format

1. header
2. data (__profd_* variables)
3. counters (__profc_* variables)
4. names (__llvm_prf_nm constant)
Demystifying the profraw header

<table>
<thead>
<tr>
<th>magic</th>
<th>__llvm_coverage_mapping[0][3] + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of</td>
<td>padding before __llvm_prf_cnts counters</td>
</tr>
<tr>
<td>size of</td>
<td>padding after __llvm_prf_data counters</td>
</tr>
<tr>
<td>size of</td>
<td>counters delta __llvm_prf_names</td>
</tr>
<tr>
<td>names begin</td>
<td>value kind last</td>
</tr>
</tbody>
</table>
Demystifying the profraw data part
Demystifying the profraw counters part
How I did it

```c
// SPDX-License-Identifier: GPL-2.0-only
#include "vmlinux.h"
#include <asm/unistd.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_core_read.h>
#include <bpf/bpf_tracing.h>

char LICENSE[] SEC("license") = "GPL";

const volatile int count = 0;

SEC("raw_tp/sys_enter")
int BPF_PROG(hook_sys_enter)
{
    bpf_printk("ciao0");

    struct trace_event_raw_sys_enter *x = (struct trace_event_raw_sys_enter *)ctx;
    if (x->id != __NR_connect)
        return 0;

    for (int i = 1; i < count; i++)
    {
        bpf_printk("ciao%d", i);
    }

    return 0;
}
```

```c
int main(int argc, char **argv)
{
    struct raw_enter *skel;
    int err;

    /* Open load and verify BPF application */
    skel = raw_enter__open();
    if (!skel) ...

    // Set the counter
    skel->rodata->count = 10;
    err = raw_enter__load(skel);
    if (err) ...

    struct trace_event_raw_sys_enter ctx = {
        .id = __NR_connect};
    struct bpf_prog_test_run_attr tattr = {
        .prog_fd = bpf_program__fd(skel->progs.hook_sys_enter),
        .ctx_in = &ctx,
        .ctx_size_in = sizeof(ctx)
    };
    err = bpf_prog_test_run_xattr(&tattr);
    cleanup:
    raw_enter__destroy(skel);
    return -err;
}
```
LLVM pass

How I did it

1. Strip the LLVM runtime profile initialization functions/ctors
2. Ensure the eBPF program is compiled with debug info
3. Fixup visibility/linkage for eBPF globals
4. Create custom eBPF sections
   - __llvm_covmap → .rodata.covmap
   - __llvm_prf_cnts → .data.prof
   - __llvm_prf_data → .rodata.prof
   - __llvm_prf_names → .rodata.profn
5. Remove the __covrec_* constant structs
   - Keep them only in the BPF ELF for llvm-cov
   - Not in the BPF ELF for loading
6. Convert the __llvm_coverage_mapping struct to:
   - 2 different global arrays (header + data)
7. Convert any __profd_* struct to:
   - 7 different global constants (ID, hash, ..., # counters, ...)
8. Annotate with the debug info all the global variables and constants
9. Keep the llvm.used in sync

```cpp
bool BPFCov::runOnModule(Module &M) {
    bool instrumented = false;
    // Bail out when missing debug info
    if (M.debug_compile_units().empty()) {
        errs() << "Missing debug info\n"
        return instrumented;
    }
    // This sequence is not random at all
    instrumented |= deleteGVarByName(M, "llvm.global_ctors");
    instrumented |= deleteFuncByName(M, "__llvm_profile_init");
    instrumented |= deleteFuncByName(M, "__llvm_profile_register_function");
    instrumented |= deleteFuncByName(M, "__llvm_profile_register_names_function");
    instrumented |= deleteFuncByName(M, "__llvm_profile_runtime_user");
    instrumented |= deleteGVarByName(M, "__llvm_profile_runtime");
    instrumented |= fixupUsedGlobals(M);
    // Stop here to avoid rewriting the profiling and coverage structs
    if (StripInitializersOnly) {
        return instrumented;
    }
    instrumented |= swapSectionWithPrefix(M, "__llvm_prf_cnts", ".data.profc");
    instrumented |= swapSectionWithPrefix(M, "__llvm_prf_names", ".rodata.profn");
    instrumented |= convertStructs(M);
    instrumented |= annotateCounters(M);
    instrumented |= swapSetSectionWithPrefix(M, "__llvm_prf_data", ".rodata.prof");
    instrumented |= swapSetSectionWithPrefix(M, "__llvm_covmap", ".rodata.covmap");
    return instrumented;
}
```
libBPFCov.so

How I did it

@llvm_coverage_mapping.0 = dso_local constant [4 x i32] [i32 0, i32 77, i32 0, i32 4], section ".rodata.covmap", align 4, !dbg !147
@llvm_coverage_mapping.1 = dso_local constant [77 x i8] c"010Jx\DA=\C9...\03", section ".rodata.covmap", align 1, !dbg !153
@prof continualizer = dso_local global [1 x i64] zeroinitializer, section ".data.procf", align 8, !dbg !158
@prof hook.sys.enter.0 = dso_local constant i64 657394334031940579, section ".rodata.profd", align 8, !dbg !164
@prof hook.sys.enter.1 = dso_local constant i64 24, section ".rodata.profd", align 8, !dbg !166
@prof hook.sys.enter.2 = dso_local constant i64 0, section ".rodata.profd", align 8, !dbg !168
@prof hook.sys.enter.3 = dso_local constant i64 0, section ".rodata.profd", align 8, !dbg !170
@prof hook.sys.enter.4 = dso_local constant i64 0, section ".rodata.profd", align 8, !dbg !172
@prof hook.sys.enter.5 = dso_local constant i32 1, section ".rodata.profd", align 4, !dbg !174
@prof hook.sys.enter.6 = dso_local constant i32 0, section ".rodata.profd", align 4, !dbg !176
@prof bpf.c.hook.sys.enter = dso_local global [3 x i64] zeroinitializer, section ".data.procf", align 8, !dbg !178
@prof bpf.c.hook.sys.enter.0 = dso_local constant i64 1956387830300976000, section ".rodata.profd", align 8, !dbg !183
@prof bpf.c.hook.sys.enter.1 = dso_local constant i64 2956750262219864, section ".rodata.profd", align 8, !dbg !185
@prof bpf.c.hook.sys.enter.2 = dso_local constant i64 8, section ".rodata.profd", align 8, !dbg !187
@prof bpf.c.hook.sys.enter.3 = dso_local constant i64 8, section ".rodata.profd", align 8, !dbg !189
@prof bpf.c.hook.sys.enter.4 = dso_local constant i64 0, section ".rodata.profd", align 8, !dbg !191
@prof bpf.c.hook.sys.enter.5 = dso_local constant i32 3, section ".rodata.profd", align 4, !dbg !193
@prof bpf.c.hook.sys.enter.6 = dso_local constant i32 0, section ".rodata.profd", align 4, !dbg !195
@llvm_prof_mm = dso_local constant [42 x i8] c"1(x\DA...\B0...\81 \03E...\13N", section ".rodata.prof", align 1, !dbg !197
How I did it

1. **bpfcov run** - run the instrumented eBPF application
   1. Detect the **eBPF globals** (__profc_*, __profd_*, ...)
   2. Detect their **custom eBPF sections**
      - .data.profc
      - .rodata.profd,
      - .rodata.profn
      - .rodata.covmap
   3. Pin them to the **BPF FS**
How I did it

1. bpfcov gen - generate the profraw from eBPF pinned maps
   1. Read the content of the **pinned eBPF maps** at:
      - /sys/fs/bpf/cov/<program>/\{profc,profld,profnn,covmap\}
   2. Dump it to a valid **profraw** file

2. bpfcov out - output coverage reports
   1. Generates **profdata** files from profraw files
   2. Merges them into a single one
   3. **HTML, JSON, LCOV** coverage reports
### Usage

#### Compilation

```bash
clang -g -O2 \  
-target bpf \  
-D__TARGET_ARCH_x86 \  
-I$(YOUR_INCLUDES) \  
-fprofile-instr-generate \  
-fcoverage-mapping \  
-emit-llvm -S \  
-c program.bpf.c \  
-o program.bpf.ll

opt -load-pass-plugin $(BUILD_DIR)/lib/libBPFCov.so \  
-passes="bpf-cov" \  
-S program.bpf.ll \  
-o program.bpf.cov.ll

llc -march=bpf -filetype=obj \  
-o cov/program.bpf.o \  
program.bpf.cov.ll

opt -load $(BUILD_DIR)/lib/libBPFCov.so \  
-strip-initializers-only -bpf-cov \  
program.bpf.ll | \  
llc -march=bpf -filetype=obj \  
-o cov/program.bpf.obj
```

#### Execution

```bash
sudo ./bpfcov run cov/program  
# Wait for it to exit  
# Or stop it with CTRL+C

sudo ./bpfcov gen --unpin cov/program

./bpfcov out \  
-o awsm_report \  
--format=html cov/program.profraw
```
Who wanna read LLVM IR for eBPF with me? 😎
Resources

- Blog post: Coverage for eBPF programs
- Writing an LLVM pass
- The Coverage Mapping format
- Dissecting the coverage mapping sample
- The encoding of the coverage mapping values: LEB128
- Demystifying the profraw format
- The functions writing the profraw file: lprofWriteData(), lprofWriteDataImpl()
- Source code (LLVM) emitting __covrec_* constants: CodeGen/CoverageMappingGen.cpp
- Calls to CoverageMappingModuleGen in LLVM: CodeGenAction::CreateASTConsumer, CodeGenModule::CodeGenModule
- Kernel patch: eBPF support for global data
- Kernel patch: libbpf: support global data/bss/rodata sections
- libbpf: arbitrarily named .rodata,* and .data,* ELF sections
- LLVM BPF target source
- How LLVM processes BPF globals
- Branch Coverage: Squeezing more out of LLVM Source-based Code Coverage by Alan Phipps
Thank you!

Questions?

— twitter.com/leodido
— github.com/leodido
— github.com/elastic/bpfcoy