bpfcov

Coverage for eBPF programs

Leonardo Di Donato - 05 Feb 2022 @ FOSDEM 22 - LLVM devroom

whoami

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**

@leodido

/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/raw_enter.bpf.c

Line	Count	Source (jump to first uncovered line)
1		// SPDX-License-Identifier: GPL-2.0
2		#include "vmlinux.h"
3		#include <asm/unistd.h>
4		#include <bpf/bpf_helpers.h>
5		#include <bpf/bpf_core_read.h>
6		#include <bpf/bpf_tracing.h>
7		
8		char LICENSE[] SEC("license") = "GPL";
9		
10		const volatile int count = 0;
11		
12		SEC("raw_tp/sys_enter")
13		int BPF_PROG(hook_sys_enter)
14	1	{
15	1	bpf_printk("ciao0");
16		
17	1	struct trace_event_raw_sys_enter *x = (struct trace_event_raw_sys_enter *)ctx;
18	1	if (x->id != __NR_connect)
		Branch (18:7): [True: 0, False: 1]
19	1	{
20	0	return 0;
21	0	}
22		
23	10	for (int i = 1; i < count; i++)
		Branch (23:19): [True: 9, False: 1]
24	9	{
25	9	bpf_printk("ciao%d", i);
26	9	}
27		
28	1	return 0;
29	1	}

Goal 🎯

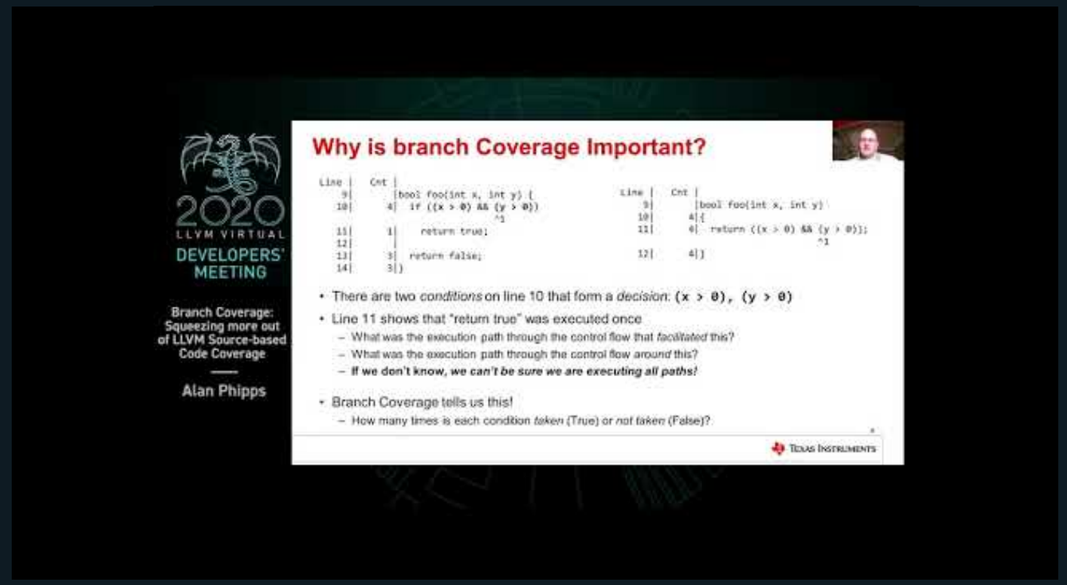
Gather **source-based code coverage** for our **eBPF applications**.

eBPF is:

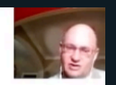
— usually written in C
— compiled via Clang to BPF ELF `.o` files
  — LLVM BPF target
— loaded through the `bpf()` syscall
— executed by the eBPF Virtual Machine in the Linux kernel

What's source-based coverage?

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



Counter Region Mapping and Instrumentation

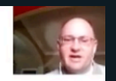


- Counters are inserted into basic blocks of generated code mapped to source

```
line  9: bool foo(int x, int y) {
          Counter1++
line 10:  if ((x > 0) && (y > 0))
                              ^Counter2++
                Counter3++
line 11:    return true;
line 12:
line 13:  return false;
line 14: }
```

- **Counter1** instrumented to track
  - Region (9:24 → 10:23)
  - Function (line 9 – `foo()`)
  - Line (line 10)
  - 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)

Why is branch Coverage Important?



```
Line |   Cnt |
   9|       |bool foo(int x, int y) {
  10|      4|  if ((x > 0) && (y > 0))
                              ^1
  11|      1|    return true;
  12|       |
  13|      3|  return false;
  14|      3|}
```

```
Line |   Cnt |
   9|       |bool foo(int x, int y)
  10|      4|{
  11|      4|  return ((x > 0) && (y > 0));
                                   ^1
  12|      4|}
```

- 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)?

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");
}
```

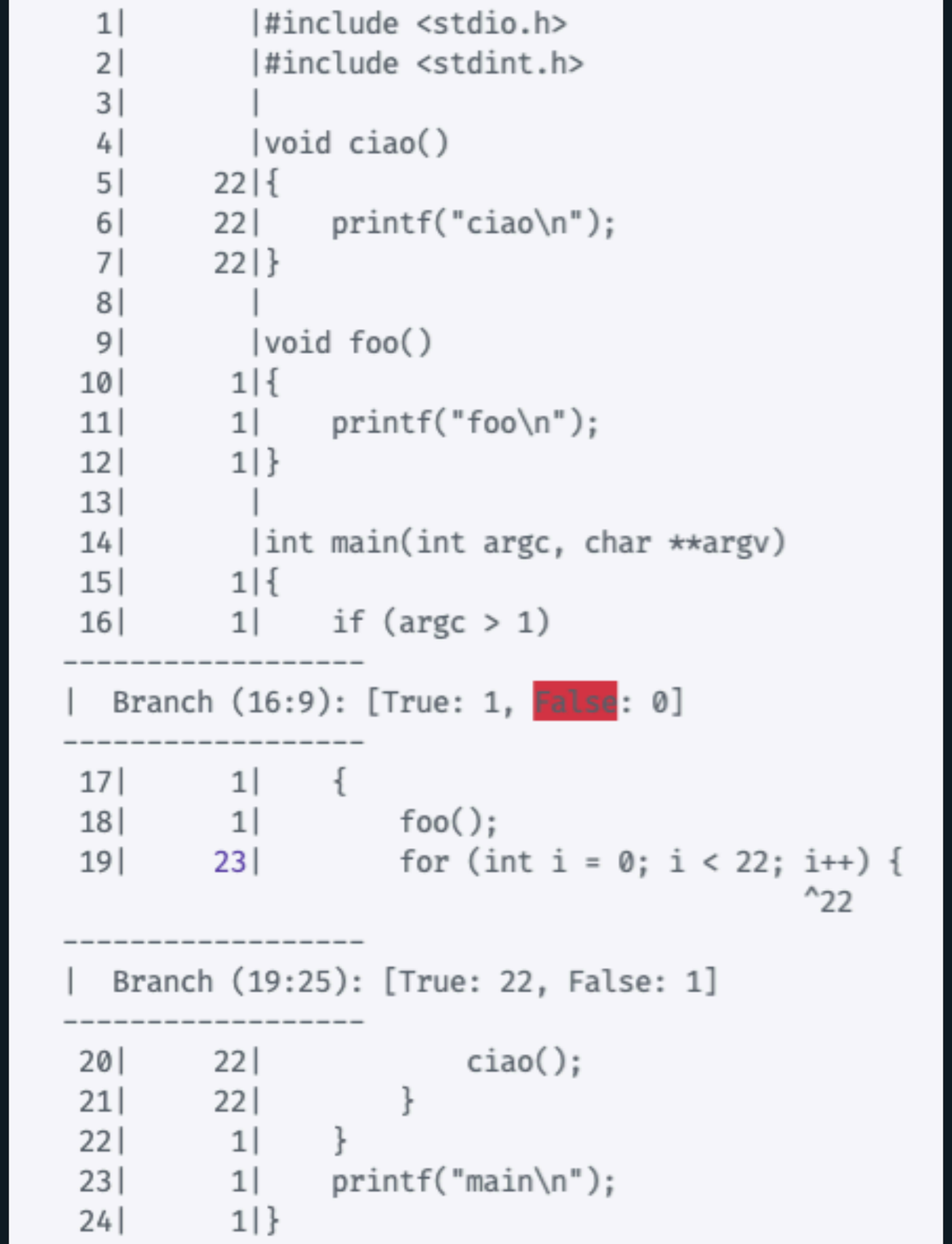
```
$ clang \
  -fprofile-instr-generate \
  -fcoverage-mapping \
  hello.c \
  -o hello


$ ./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
```

[1] for more details visit the LLVM docs



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

-fprofile-instr-generate

Instruments the program functions to collect execution counts

```
@__profc_ciao = private global [1 x i64] zeroinitializer, section "__llvm_prf_cnts", align 8
@__profd_ciao = private global { i64, i64, i64*, i8*, i8*, i32, [2 x i16] } {
    i64 -1479480177954886802,
    i64 0,
    i64* getelementptr inbounds ([1 x i64], [1 x i64]* @__profc_ciao, i32 0, i32 0),
    i8* bitcast (void ()* @ciao to i8*), i8* null, i32 1, [2 x i16] zeroinitializer
    }, section "__llvm_prf_data", align 8
@__profc_foo = private global [1 x i64] zeroinitializer, section "__llvm_prf_cnts", align 8
@__profd_foo = private global { i64, i64, i64*, i8*, i8*, i32, [2 x i16] } {
    i64 6699318081062747564,
    i64 0,
    i64* getelementptr inbounds ([1 x i64], [1 x i64]* @__profc_foo, i32 0, i32 0),
    i8* bitcast (void ()* @foo to i8*), i8* null, i32 1, [2 x i16] zeroinitializer
    }, section "__llvm_prf_data", align 8
@__profc_main = private global [3 x i64] zeroinitializer, section "__llvm_prf_cnts", align 8
@__profd_main = private global { i64, i64, i64*, i8*, i8*, i32, [2 x i16] } {
    i64 -2624081020897602054,
    i64 717562688593,
    i64* getelementptr inbounds ([3 x i64], [3 x i64]* @__profc_main, i32 0, i32 0),
    i8* bitcast (i32 (i32, i8**)* @main to i8*), i8* null, i32 3, [2 x i16] zeroinitializer
    }, section "__llvm_prf_data", align 8
@__llvm_prf_nm = private constant [15 x i8] c"\0D\00ciao\01foo\01main", section "__llvm_prf_names", align 1
```

ciao()

foo()

main()

```
define dso_local void @foo() #0 !dbg !15 {
  %1 = load i64, i64* getelementptr inbounds ([1 x i64], [1 x i64]* @__profc_foo, 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]* @__profc_foo, i64 0, i64 0), align 8, !dbg !16
  %3 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([5 x i8], [5 x i8]* @.str.1, 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]* @__profc_main, i64 0, i64 0), align 8, !dbg !30
  %8 = add i64 %7, 1, !dbg !30
  store i64 %8, i64* getelementptr inbounds ([3 x i64], [3 x i64]* @__profc_main, 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 i1 %10, label %11, label %24, !dbg !34

11:                                               ; preds = %2
  %12 = load i64, i64* getelementptr inbounds ([3 x i64], [3 x i64]* @__profc_main, i64 0, i64 1), align 8, !dbg !34
  %13 = add i64 %12, 1, !dbg !34
  store i64 %13, i64* getelementptr inbounds ([3 x i64], [3 x i64]* @__profc_main, 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

...
```

foo()'s entry: increment by 1

main()'s entry: increment by 1

main()'s if: increment by 1

-fcoverage-mapping

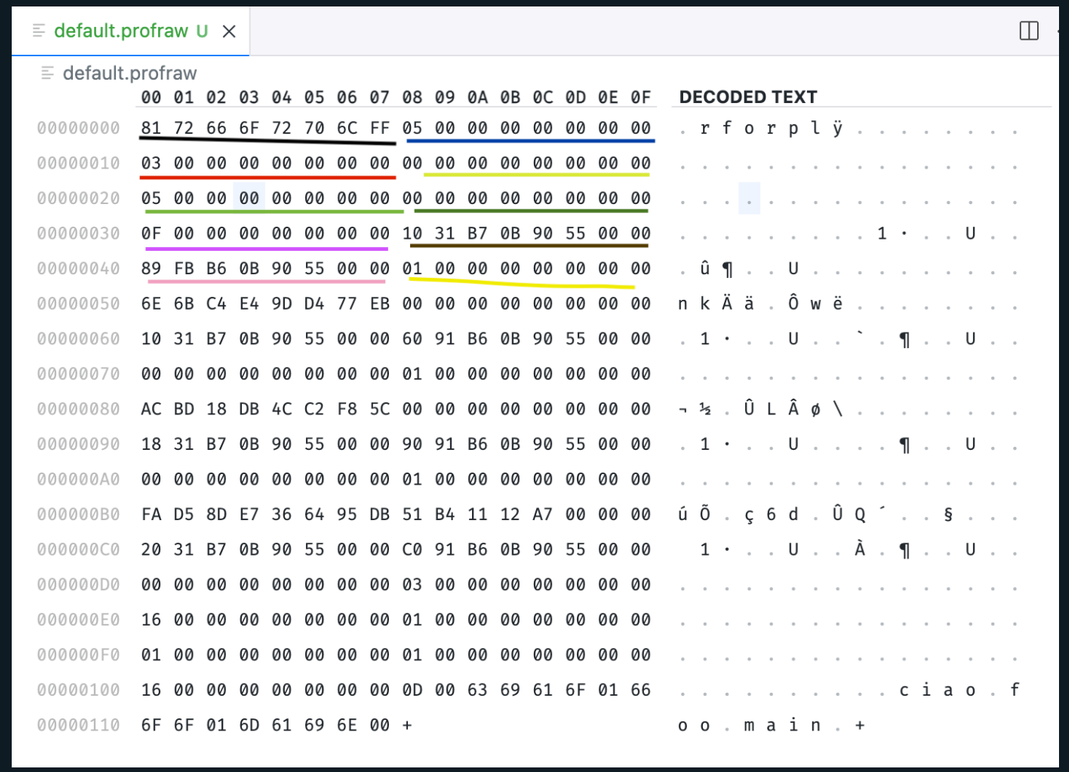
Generate coverage mappings

```
@__covrec_EB77D49DE4C46B6Eu = linkonce_odr hidden constant <{ i64, i32, i64, i64, [9 x i8] }> <{          ciao()
    i64 -1479480177954886802,
    i32 9,
    i64 0,
    i64 -6624215419316037574,
    [9 x i8] c"\01\00\00\01\01\05\01\02\02"
    }>, section "__llvm_covfun", comdat, align 8
@__covrec_5CF8C24CDB18BDACu = linkonce_odr hidden constant <{ i64, i32, i64, i64, [9 x i8] }> <{          foo()
    i64 6699318081062747564,
    i32 9,
    i64 0,
    i64 -6624215419316037574,
    [9 x i8] c"\01\00\00\01\01\0A\01\02\02"
    }>, section "__llvm_covfun", comdat, align 8
@__covrec_DB956436E78DD5FAu = linkonce_odr hidden constant <{ i64, i32, i64, i64, [70 x i8] }> <{          main()
    i64 -2624081020897602054,
    i32 70, i64 717562688593,
    i64 -6624215419316037574,
    [70 x i8] c"\01\00\02\01\05\05\09\0A\01\0F\01\09\02\01\01\09\00\11 \05\...\1F \09...\0A"
    }>, section "__llvm_covfun", comdat, align 8
@__llvm_coverage_mapping = private constant { { i32, i32, i32, i32 }, [72 x i8] } {
    { i32, i32, i32, i32 } { i32 0, i32 72, i32 0, i32 4 },
    [72 x i8] c"\01E\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



magic	__llvm_coverage_mapping[0][3] + 1
size of __llvm_prf_cnts	padding before counters
size of __llvm_prf_data	padding after counters
size of __llvm_prf_names	counters delta
names begin	value kind last

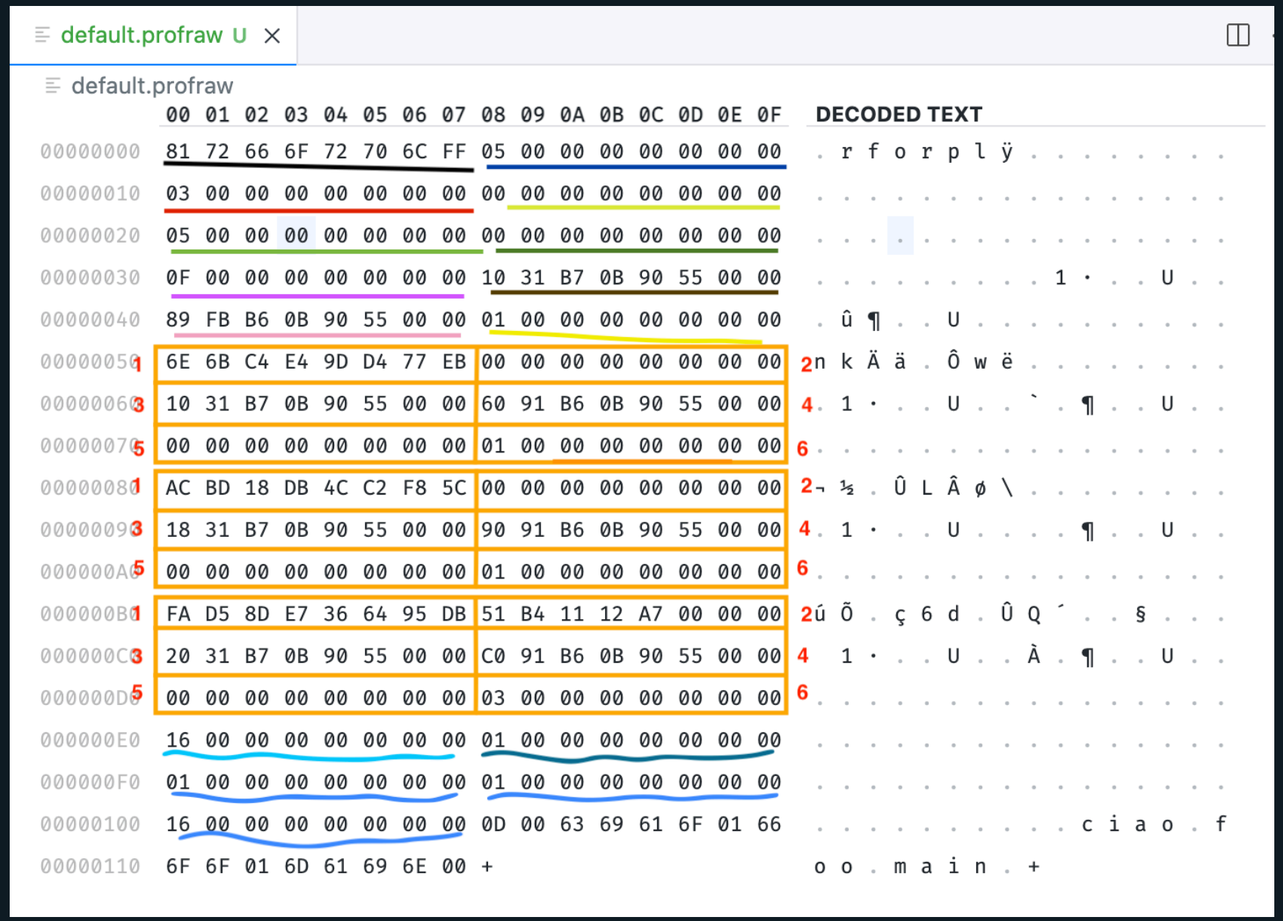
Demystifying the profraw data part

default.profraw U

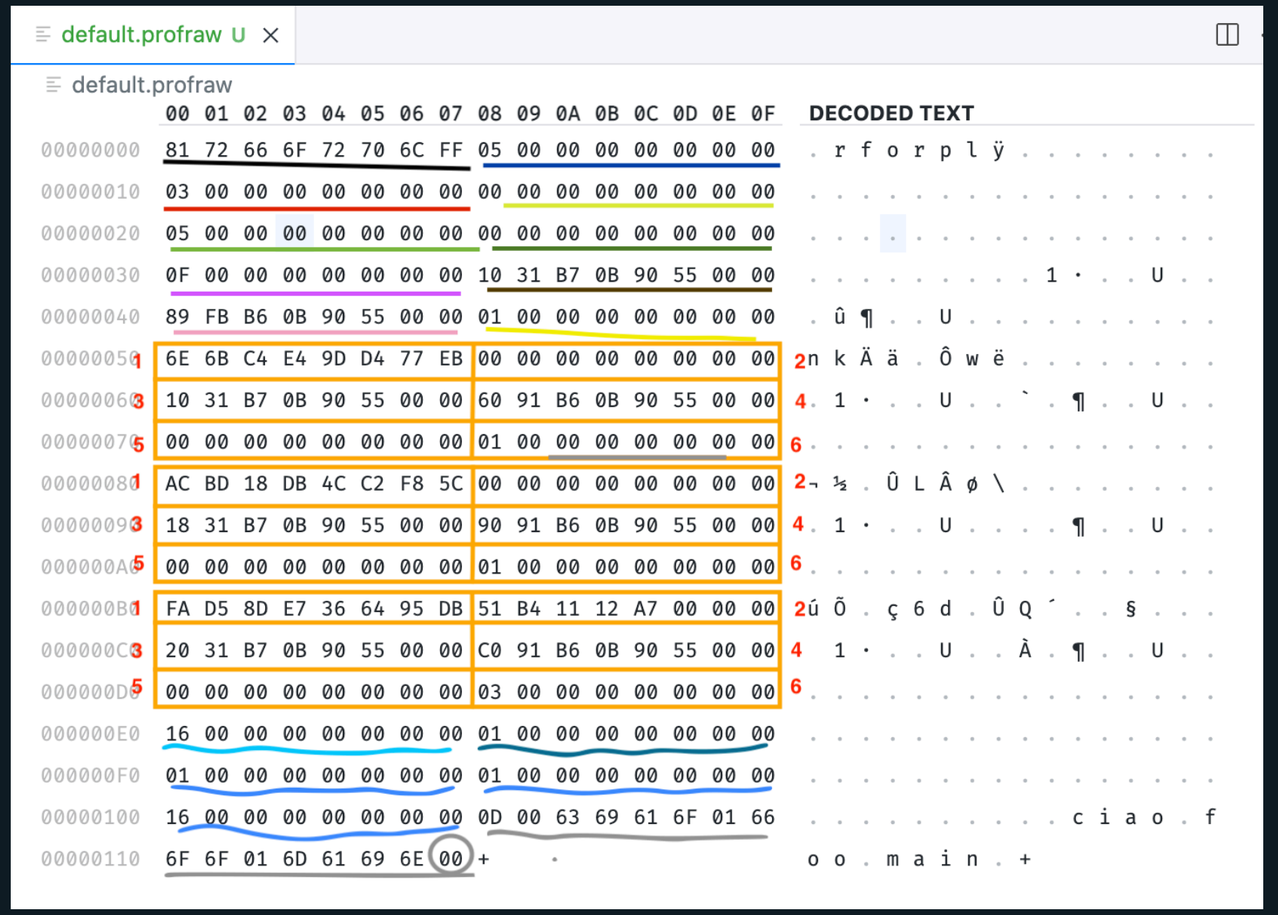
default.profraw

	00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F	DECODED TEXT
00000000	81 72 66 6F 72 70 6C FF 05 00 00 00 00 00 00 00	. r f o r p l ÿ . . . . .
00000010	03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
00000020	05 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	. . . . .
00000030	0F 00 00 00 00 00 00 00 10 31 B7 0B 90 55 00 00	. . . . . 1 · . . U . .
00000040	89 FB B6 0B 90 55 00 00 01 00 00 00 00 00 00 00	. û ¶ . . U . . . .
00000050 1	6E 6B C4 E4 9D D4 77 EB 00 00 00 00 00 00 00 00 2	n k Ä ä . Ô w ë . . . . .
00000060 3	10 31 B7 0B 90 55 00 00 60 91 B6 0B 90 55 00 00 4	. 1 · . . U . . ` . ¶ . . U . .
00000070 5	00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 6	. . . . .
00000080 1	AC BD 18 DB 4C C2 F8 5C 00 00 00 00 00 00 00 00 2	¬ ½ . Û L Â ø \ . . . . .
00000090 3	18 31 B7 0B 90 55 00 00 90 91 B6 0B 90 55 00 00 4	. 1 · . . U . . . . ¶ . . U . .
000000A0 5	00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00 6	. . . . .
000000B0 1	FA D5 8D E7 36 64 95 DB 51 B4 11 12 A7 00 00 00 2	ú Õ . ç 6 d . Û Q ´ . . § . . .
000000C0 3	20 31 B7 0B 90 55 00 00 C0 91 B6 0B 90 55 00 00 4	1 · . . U . . À . ¶ . . U . .
000000D0 5	00 00 00 00 00 00 00 00 03 00 00 00 00 00 00 00 6	. . . . .
000000E0	16 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00	. . . . .
000000F0	01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00	. . . . .
00000100	16 00 00 00 00 00 00 00 0D 00 63 69 61 6F 01 66	. . . . . c i a o . f
00000110	6F 6F 01 6D 61 69 6E 00 +	o o . m a i n . +

Demystifying the profraw counters part



Demystifying the profraw names part



Patching LLVM IR for eBPF coverage

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
// SPDX-License-Identifier: GPL-2.0-only
#include <asm/unistd.h>
#include <bpf/bpf.h>
#include "commons.c"
#include "raw_enter.skel.h"

...

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.profc`
   — `__llvm_prf_data` → `.rodata.profd`
   — `__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

```
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 |= swapSectionWithPrefix(M, "__llvm_prf_data", ".rodata.profd");
    instrumented |= swapSectionWithPrefix(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 !47
@__llvm_coverage_mapping.1 = dso_local constant [77 x i8] c"\01OJx\DA=\C9\...\03", section ".rodata.covmap", align 1, !dbg !53
@__profc_hook_sys_enter = dso_local global [1 x i64] zeroinitializer, section ".data.profc", align 8, !dbg !58
@__profd_hook_sys_enter.0 = dso_local constant i64 6573943340031040579, section ".rodata.profd", align 8, !dbg !64
@__profd_hook_sys_enter.1 = dso_local constant i64 24, section ".rodata.profd", align 8, !dbg !66
@__profd_hook_sys_enter.2 = dso_local constant i64 0, section ".rodata.profd", align 8, !dbg !68
@__profd_hook_sys_enter.3 = dso_local constant i64 0, section ".rodata.profd", align 8, !dbg !70
@__profd_hook_sys_enter.4 = dso_local constant i64 0, section ".rodata.profd", align 8, !dbg !72
@__profd_hook_sys_enter.5 = dso_local constant i32 1, section ".rodata.profd", align 4, !dbg !74
@__profd_hook_sys_enter.6 = dso_local constant i32 0, section ".rodata.profd", align 4, !dbg !76
@__profc_raw_enter.bpf.c_____hook_sys_enter = dso_local global [3 x i64] zeroinitializer, section ".data.profc", align 8, !dbg !78
@__profd_raw_enter.bpf.c_____hook_sys_enter.0 = dso_local constant i64 1956387830300976000, section ".rodata.profd", align 8, !dbg !83
@__profd_raw_enter.bpf.c_____hook_sys_enter.1 = dso_local constant i64 2956750262219864, section ".rodata.profd", align 8, !dbg !85
@__profd_raw_enter.bpf.c_____hook_sys_enter.2 = dso_local constant i64 8, section ".rodata.profd", align 8, !dbg !87
@__profd_raw_enter.bpf.c_____hook_sys_enter.3 = dso_local constant i64 0, section ".rodata.profd", align 8, !dbg !89
@__profd_raw_enter.bpf.c_____hook_sys_enter.4 = dso_local constant i64 0, section ".rodata.profd", align 8, !dbg !91
@__profd_raw_enter.bpf.c_____hook_sys_enter.5 = dso_local constant i32 3, section ".rodata.profd", align 4, !dbg !93
@__profd_raw_enter.bpf.c_____hook_sys_enter.6 = dso_local constant i32 0, section ".rodata.profd", align 4, !dbg !95
@__llvm_prf_nm = dso_local constant [42 x i8] c"1(x\DA...\B0...\81 \03E...\13N", section ".rodata.profn", align 1, !dbg !97
```

BPF_PROG(hook_sys_enter){}

hook_sys_enter(){}

./bpfcov run ...

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**

./bpfcov gen|out ...

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,profd,profn,covmap}`
    2. Dump it to 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

```
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

```
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
```

Demo

Who wanna read LLVM IR for eBPF with me? 😎

```
1|      |// SPDX-License-Identifier: GPL-2.0
2|      |#include "vmlinux.h"
3|      |#include <asm/unistd.h>
4|      |#include <bpf/bpf_helpers.h>
5|      |#include <bpf/bpf_core_read.h>
6|      |#include <bpf/bpf_tracing.h>
7|      |
8|      |char LICENSE[] SEC("license") = "GPL";
9|      |
10|     |const volatile int count = 0;
11|     |
12|     |SEC("raw_tp/sys_enter")
13|     |int BPF_PROG(hook_sys_enter)
14|    1|{
15|    1|  bpf_printk("ciao0");
16|     |
17|    1|  struct trace_event_raw_sys_enter *x = (struct trace_event_raw_sys_enter *)ctx;
18|    1|  if (x->id != __NR_connect)
19|    1|  {
20|    0|    return 0;
21|    0|  }
22|     |
23|   10|  for (int i = 1; i < count; i +)
            ^1                    ^9
24|    9|  {
25|    9|    bpf_printk("ciao%d", i);
26|    9|  }
27|     |
```

```
94
95   8  __u32 pid = bpf_get_current_pid_tgid() >> 32;
96   8  int is_stack = 0;
97
98   8  is_stack = (vma->vm_start <= vma->vm_mm->start_stack && vma->vm_end >= vma->vm_mm->start_stack);
       Branch (98:14): [True: 8, False: 0]
       Branch (98:58): [True: 2, False: 6]
99
100  8  if (is_stack && monitored_pid == pid) {
       Branch (100:6): [True: 2, False: 6]
       Branch (100:18): [True: 2, False: 0]
101  2    mprotect_count++;
102  2    ret = -EPERM;
103  2  }
104
105  8  return ret;
106  8}
107
108    SEC("lsm.s/bprm_committed_creds")
109    int BPF_PROG(test_void_hook, struct linux_binprm *bprm)
110  2{
111  2  __u32 pid = bpf_get_current_pid_tgid() >> 32;
112  2  int ret, inner_map, inner_map;
```

```
151|     2|                *value = 0;
152|     2|  }
153|      |
154|     2|  return 0;
155|     2|}
156|      |
157|      |SEC("lsm/task_free") /* lsm/ is ok, lsm.s/ fails */
158|      |int BPF_PROG(test_task_free, struct task_struct *task)
159|     2|{
160|     2|  return 0;
161|     2|}
162|      |
163|      |int copy_test = 0;
164|      |
165|      |SEC("fentry.s/__x64_sys_setdomainname")
166|      |int BPF_PROG(test_sys_setdomainname, struct pt_regs *regs)
167|     6|{
168|     6|  void *ptr = (void *)PT_REGS_PARM1(regs);
169|     6|  int len = PT_REGS_PARM2(regs);
170|     6|  int buf = 0;
171|     6|  long ret;
172|      |
173|     6|  ret = bpf_copy_from_user(&buf, sizeof(buf), ptr);
174|     6|  if (len == -2 && ret == 0 && buf == 1234)
                          ^2          ^2
175|     2|          copy_test +;
176|     6|  if (len == -3 && ret == -EFAULT)
                          ^2          ^2
177|     6|#include <bpf/bpf_tracing.h>
```

```
7
8    char LICENSE[] SEC("license") = "GPL";
9
10   const volatile int count = 0;
11
12   SEC("raw_tp/sys_enter")
13   int BPF_PROG(hook_sys_enter)
14   1{
15   1  bpf_printk("ciao0");
16
17   1  struct trace_event_raw_sys_enter *x = (struct trace_event_raw_sys_enter *)ctx;
18   1  if (x->id != __NR_connect)
       Branch (18:7): [True: 0, False: 1]
19   1  {
20   0    return 0;
21   0  }
22
```

```
DA:11,20
DA:13,20
DA:14,20
DA:15,20
DA:16,20
DA:20,20
DA:21,20
DA:23,20
DA:24,20
DA:25,20
DA:26,20
BRF:0
BFH:0
LF:12
LH:12
end_of_record
SF:/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/lsm.bpf.c
FN:91,lsm.bpf.c:____test_int_hook
FN:110,lsm.bpf.c:____test_void_hook
```

LCOV - code coverage report

		Hit	Total	Coverage
Current view:	top level - src			
Test:	all.info	Lines:	91	98	92.9 %
Date:	2022-01-12 16:29:29	Functions:	7	7	100.0 %
Legend:	Rating: low: < 75 % medium: >= 75 % high: >= 90 %				

Filename	Line Coverage (show details)		Functions	
fentry.bpf.c	100.0 %	12 / 12	100.0 %	2 / 2
lsm.bpf.c	93.2 %	68 / 73	100.0 %	4 / 4
raw_enter.bpf.c	84.6 %	11 / 13	100.0 %	1 / 1

Generated by: LCOV version 1.15

Coverage Report

Created: 2022-01-12 15:57

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
fentry.bpf.c	100.00% (2/2)	100.00% (12/12)	100.00% (2/2)	- (0/0)
lsm.bpf.c	100.00% (4/4)	92.96% (66/71)	90.38% (47/52)	60.87% (28/46)
raw_enter.bpf.c	100.00% (1/1)	84.62% (11/13)	85.71% (6/7)	75.00% (3/4)
Totals	**100.00% (7/7)**	**92.71% (89/96)**	**90.16% (55/61)**	**62.00% (31/50)**

Generated by llvm-cov -- llvm version 12.0.1

{"data":[{"files":[{"filename":"/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/fentry.bpf.c","summary":{"functions":{"count":2,"covered":2,"percent":100},"instantiations":{"count":2,"covered":2,"percent":100},"lines":{"count":12,"covered":12,"percent":100},"regions":{"count":2,"covered":2,"notcovered":0,"percent":100},"branches":{"count":0,"covered":0,"notcovered":0,"percent":0}}},{"filename":"/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/lsm.bpf.c","summary":{"functions":{"count":4,"covered":4,"percent":100},"instantiations":{"count":4,"covered":4,"percent":100},"lines":{"count":71,"covered":66,"percent":92.9577},"regions":{"count":52,"covered":47,"notcovered":5,"percent":90.3846},"branches":{"count":46,"covered":28,"notcovered":18,"percent":60.8696}}},{"filename":"/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/raw_enter.bpf.c","summary":{"functions":{"count":1,"covered":1,"percent":100},"instantiations":{"count":1,"covered":1,"percent":100},"lines":{"count":13,"covered":11,"percent":84.6154},"regions":{"count":7,"covered":6,"notcovered":1,"percent":85.7143},"branches":{"count":4,"covered":3,"notcovered":1,"percent":75}}},{"filename":"/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/lsm.bpf.c","segments":[[91,1,8,true,true,false],[92,6,8,true,true,false],[92,14,8,true,false,false],[92,15,93,1,0,0,0,1],[93,3,93,13,0,0,0,1],[95,2,106,2,0,0,0,0],[97,1,97,1,0,0,0,2],[98,14,98,54,8,0,0,0],[98,58,98,96,8,0,0,0],[99,1,99,1,0,0,0,2],[100,6,100,14,8,0,0,0],[100,6,100,38,8,0,0,0],[100,18,100,38,2,0,0,0],[100,39,100,40,2,0,0,3],[100,40,103,3,2,0,0,0],[102,10,102,15,2,0,1,1],[104,1,104,1,0,0,0,2],[12,15,12,16,2,1,0,0]]},{"filename":"/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/lsm.bpf.c","segments":[[176,27,176,33,2,0,1,1],[167,1,181,2,6,0,0,0],[172,1,172,1,0,0,0,2],[174,6,174,27,6,0,0,0],[174,31,174,42,2,0,0,0],[174,43,175,3,2,0,0,3],[175,3,175,14,2,0,0,0],[176,6,176,15,6,0,0,0],[176,6,176,33,6,0,0,0],[176,19,176,33,2,0,0,0],[177,3,177,14,2,0,0,0],[178,6,178,15,6,0,0,0],[178,6,178,33,6,0,0,0],[178,19,178,33,2,0,0,0],[178,27,178,33,2,0,2,1],[179,3,179,14,2,0,0,0],[13,16,13,18,2,1,0,0],[13,16,13,18,2,2,0,0]]},{"branches":[],"filenames":["/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/lsm.bpf.c","/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/lsm.bpf.c"],"source_region":[178,27,178,33,2,0,2,1],"target_regions":[[167,1,181,2,6,0,0,0],[172,1,172,1,0,0,0,2],[174,6,174,15,6,0,0,0],[174,6,174,27,6,0,0,0],[174,6,174,42,6,0,0,0],[174,19,174,27,2,0,0,0],[174,31,174,42,2,0,0,0],[174,43,175,3,2,0,0,3],[175,3,175,14,2,0,0,0],[176,6,176,15,6,0,0,0],[176,6,176,33,6,0,0,0],[176,19,176,33,2,0,0,0],[177,3,177,14,2,0,0,0],[178,6,178,15,6,0,0,0],[178,6,178,33,6,0,0,0],[178,19,178,33,2,0,0,0],[178,27,178,33,2,0,2,1],[179,3,179,14,2,0,0,0],[13,16,13,18,2,1,0,0],[13,16,13,18,2,2,0,0]]}],"filename":"/home/leodido/workspace/github.com/leodido/bpfcov/examples/src/lsm.bpf.c","segments":[[12,15,2,true,true,false],[12,16,0,false,false,false],[13,16,4,true,true,false],[13,18,0,false,false,false],[91,1,8,true,true,false],[92,6,8,true,true,false],[92,14,8,true,false,false],[92,15,0,true,false,true],[93,3,0,true,true,false],[93,13,8,true,false,true],[94,1,0,false,true,false],[94,1,8,true,false,true],[95,2,8,true,true,false],[97,1,0,false,true,false],[97,1,8,true,false,false],[98,14,8,true,true,false],[98,54,8,true,false,false],[98,58,8,true,true,false],[98,96,8,true,false,false],[99,1,0,false,true,false],[99,1,8,true,false,false],[100,6,8,true,true,false],[100,14,8,true,false,false],[100,18,2,true,true,false],[100,38,0,true,false,false],[100,39,2,true,false,true],[100,40,2,true,true,false],[102,10,2,true,true,false],[102,15,2,true,false,false],[103,3,8,true,false,false],[104,1,0,false,true,false],[104,1,8,true,false,false],[106,2,0,false,false,false],[110,1,2,true,true,false],[116,1,0,false,true,false],[116,1,2,true,false,false],[117,6,2,true,false,false],[117,26,2,true,false,false],[118,3,2,true,true,false],[118,15,2,true,false,false],[119,1,0,false,true,false],[119,1,2,true,false,false],[122,1,0,false,true,false],[122,1,2,true,false,false],[124,6,2,true,true,false],[124,11,2,true,false,false],[125,3,2,true,true,false],[125,13,2,true,false,false],[127,6,2,true,true,false],[127,11,2,true,false,false],[127,12,0,true,false,true],[128,3,0,true,true,false],[128,13,2,true,false,false],[130,6,2,true,true,false],[130,11,2,true,false,false],[130,12,0,true,false,true],[131,3,8,true,true,false],[131,13,2,true,false,false],[133,6,2,true,true,false],[133,11,2,true,false,false],[134,3,2,true,true,false],[134,13,2,true,false,false],[136,6,2,true,true,false],[136,11,2,true,false,false],[136,12,0,true,false,true],[137,3,0,true,true,false],[137,13,2,true,false,false],[139,6,2,true,true,false],[139,11,2,true,false,false],[139,12,0,true,false,true],[140,3,0,true,true,false],[140,13,2,true,false,false],[142,6,2,true,true,false],[142,15,2,true,false,false],[144,7,2,true,true,false],[144,12,2,true,false,false],[145,4,2,true,true,false],[145,14,2,true,false,false],[148,6,2,true,true,false],[148,15,2,true,false,false],[148,17,2,true,true,false],[150,7,2,true,true,false],[150,12,2,true,false,false],[151,4,2,true,true,false],[151,14,2,true,false,false],[153,1,0,false,false,false],[153,1,2,true,false,false],[155,2,0,false,false,false],[159,1,2,true,true,false]]}]}

Test:	all.info	Lines:	11	13	84.6 %
Date:	2022-01-12 16:29:29	Functions:	1	1	100.0 %
Legend:	Lines: hit not hit				

Line data	Source code
1	: // SPDX-License-Identifier: GPL-2.0
2	: #include "vmlinux.h"
3	: #include <asm/unistd.h>
4	: #include <bpf/bpf_helpers.h>
5	: #include <bpf/bpf_core_read.h>
6	: #include <bpf/bpf_tracing.h>
7	:
8	: char LICENSE[] SEC("license") = "GPL";
9	:
10	: const volatile int count = 0;
11	:
12	: SEC("raw_tp/sys_enter")
13	: int BPF_PROG(hook_sys_enter)
14	1 : {
15	1 : bpf_printk("ciao0");
16	:
17	1 : struct trace_event_raw_sys_enter *x = (struct trace_event_raw_sys_enter *)ctx;
18	1 : if (x->id != __NR_connect)
19	1 : {
20	0 : return 0;
21	0 : }
22	:
23	10 : for (int i = 1; i < count; i++)
24	9 : {
25	9 : bpf_printk("ciao%d", i);
26	9 : }
27	:

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: arbitrarly 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/bpfcov