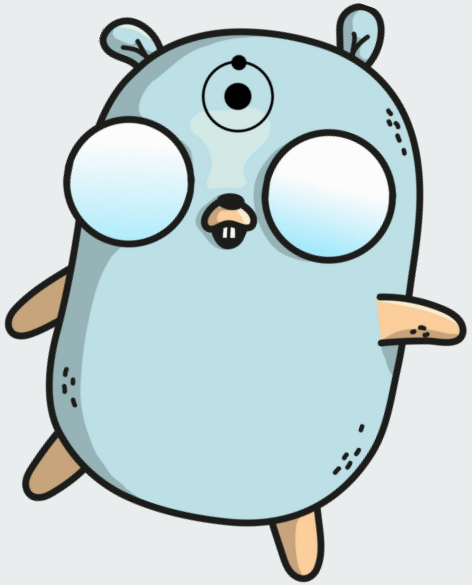
Tracing Go Programs with eBPF!

Grant Seltzer Richman
(he/him/his)



Slides/code available at grant.pizza/talks/gophercon2020


Who am I?

Who am I?

- I interned at Red Hat working on container security tools
- I worked at Capsule8 where I really dove into Linux internals
- I’m currently a security engineer at Oscar Health
- Using Go my whole career

What is tracing?

By using many features of the operating system we can inspect the activity of running programs.

We can make sure programs are doing what they say they're doing.

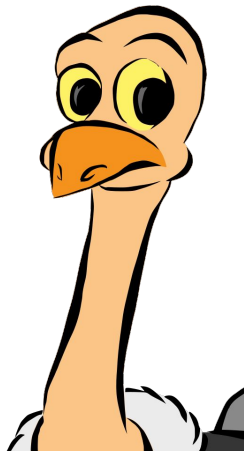
We can see why programs aren't working properly.

We can monitor performance.

For example, strace

- Monitors processes for system calls, signals, changes to process state.
- Built using ptrace, does not use eBPF

```
[*] strace go
execve("/usr/bin/go", ["go"], 0x7ffd34e27260 /* 50 vars */) = 0
brk(NULL)                                       = 0x12a9000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffd71ab3370) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=83225, ...}) = 0
mmap(NULL, 83225, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f8714d8f000
close(3)                                        = 0
openat(AT_FDCWD, "/lib64/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\240\201\0\0\0\0\0\0"...,
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0]\330\300\236^xS1\202\323~\21\0302 w"..
fstat(3, {st_mode=S_IFREG|0755, st_size=317728, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f87
pread64(3, "\4\0\0\0\24\0\0\0\3\0\0\0GNU\0]\330\300\236^xS1\202\323~\21\0302 w"..
mmap(NULL, 135600, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8714d6b000
mmap(0x7f8714d72000, 65536, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWR
```



Another example, execsnoop

- Monitors the system for newly exec'ed programs
- Built using eBPF!

```
Tracing exec()s. Ctrl-C to end.
Instrumenting do_execve
   PID    PPID ARGS
 82845   82843 cat -v trace_pipe
 82844   82840 gawk -v o=1 -v opt_name=0 -v name= -v opt_duration=0 [...]
 82848   82847 git config --get oh-my-zsh.hide-status
 82850   82849 git symbolic-ref HEAD
 82852   82851 git rev-parse --short HEAD
 82853   82690 go
 82859   82690 /usr/libexec/vte-urlencode-cwd
 82862   82861 git config --get oh-my-zsh.hide-status
 82864   82863 git symbolic-ref HEAD
 82866   82865 git rev-parse --short HEAD
 82873   82690 flameshot gui
```

eBPF is used to build tracing tools

- A virtual machine that runs inside the Linux kernel.
- Let’s you write C code which is executed when triggered by events.
- You can define the event triggers - system calls being executed, network packets arriving, functions in programs being run.

- In their simplest form, eBPF programs can just print a log message.
- However, you they have many helper functions and access to a lot of data, so you can do some really cool things with them!

```
1  #include <uapi/linux/ptrace.h>
2
3  int log_event_triggering(struct pt_regs *ctx) {
4      bpf_trace_printk("The event was triggered!")
5      return 0;
6  }
```

uprobes

- Lets you trigger eBPF programs based on source code functions being called in running processes.
- Attaches to the symbols in binaries.

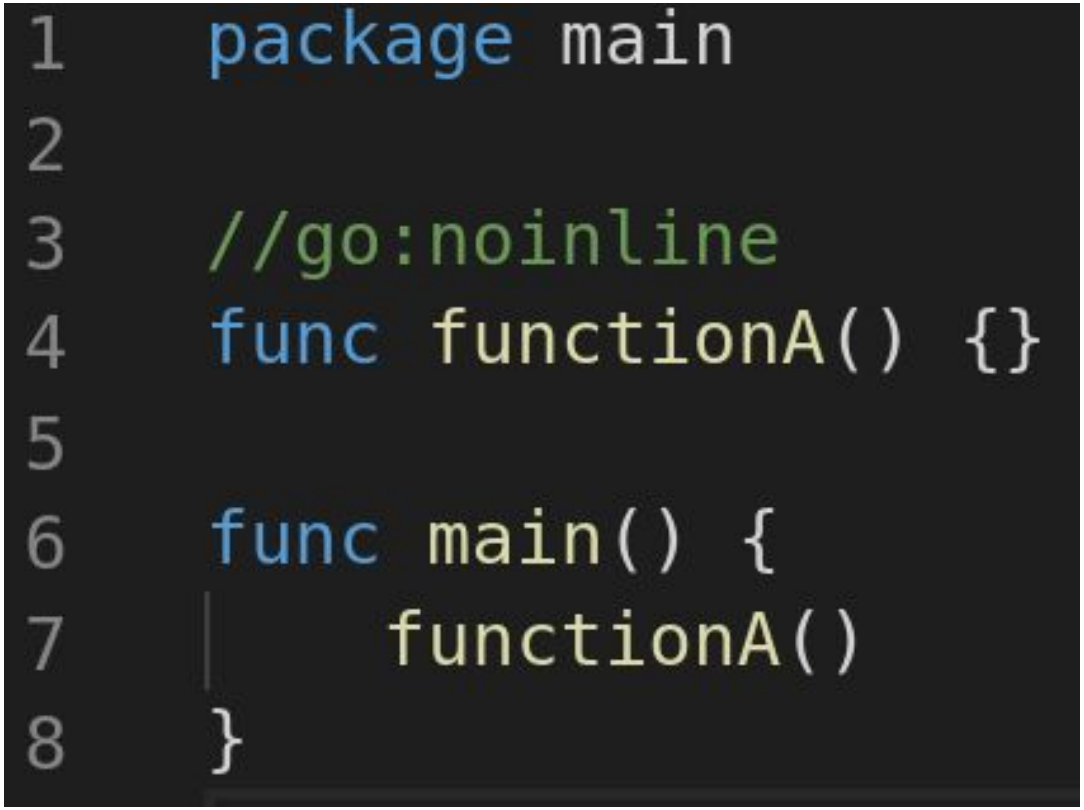
```
[*] nm ./demo | grep "main"
00000000004586b0 T main.functionA
00000000004cd0b0 D main..inittask
00000000004586c0 T main.main
000000000042c070 T runtime.main
0000000000450bc0 T runtime.main.func1
0000000000450c10 T runtime.main.func2
00000000004d04f0 B runtime.main_init_done
0000000000486060 R runtime.mainPC
00000000004f99cf B runtime.mainStarted
```

BCC (BPF Compiler Collection)

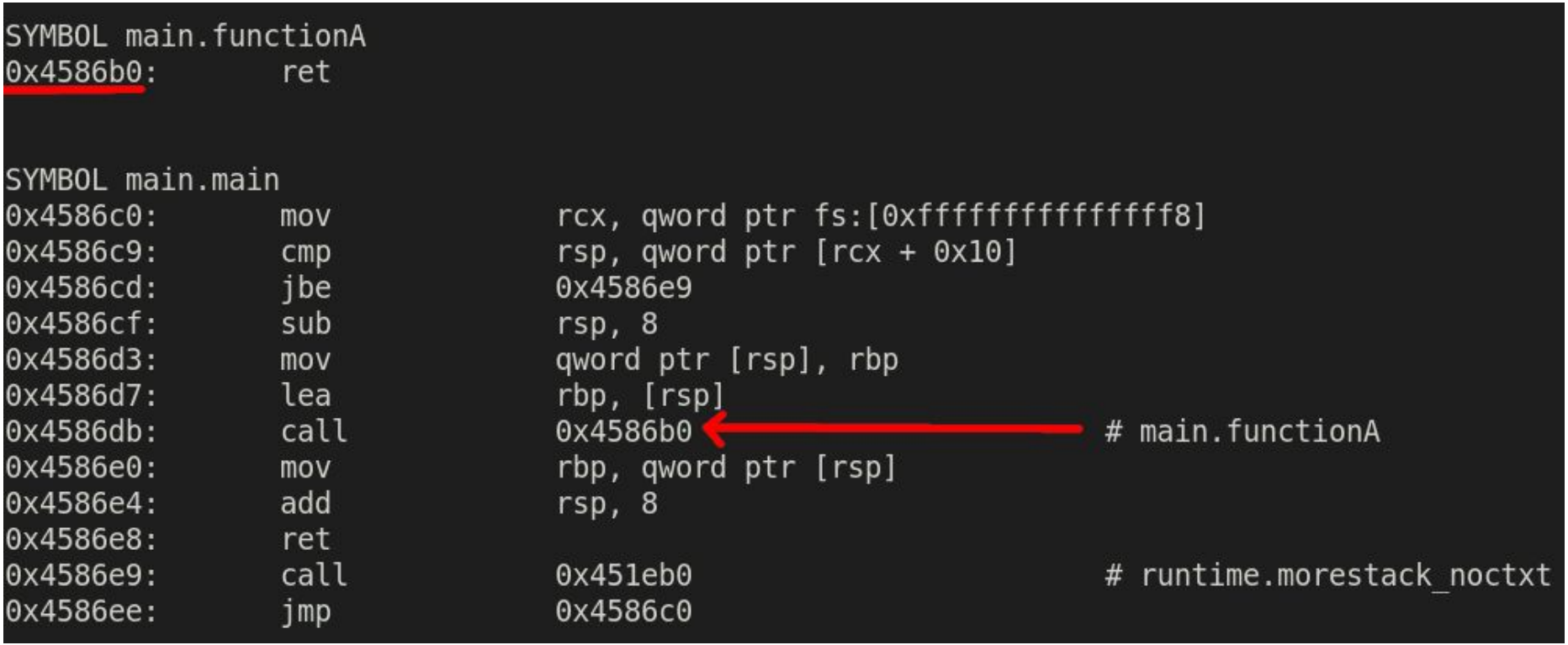
- A compiler/library for eBPF code.
- You can invoke it directly from Go.
- Makes it really easy to load, and orchestrate your eBPF code.

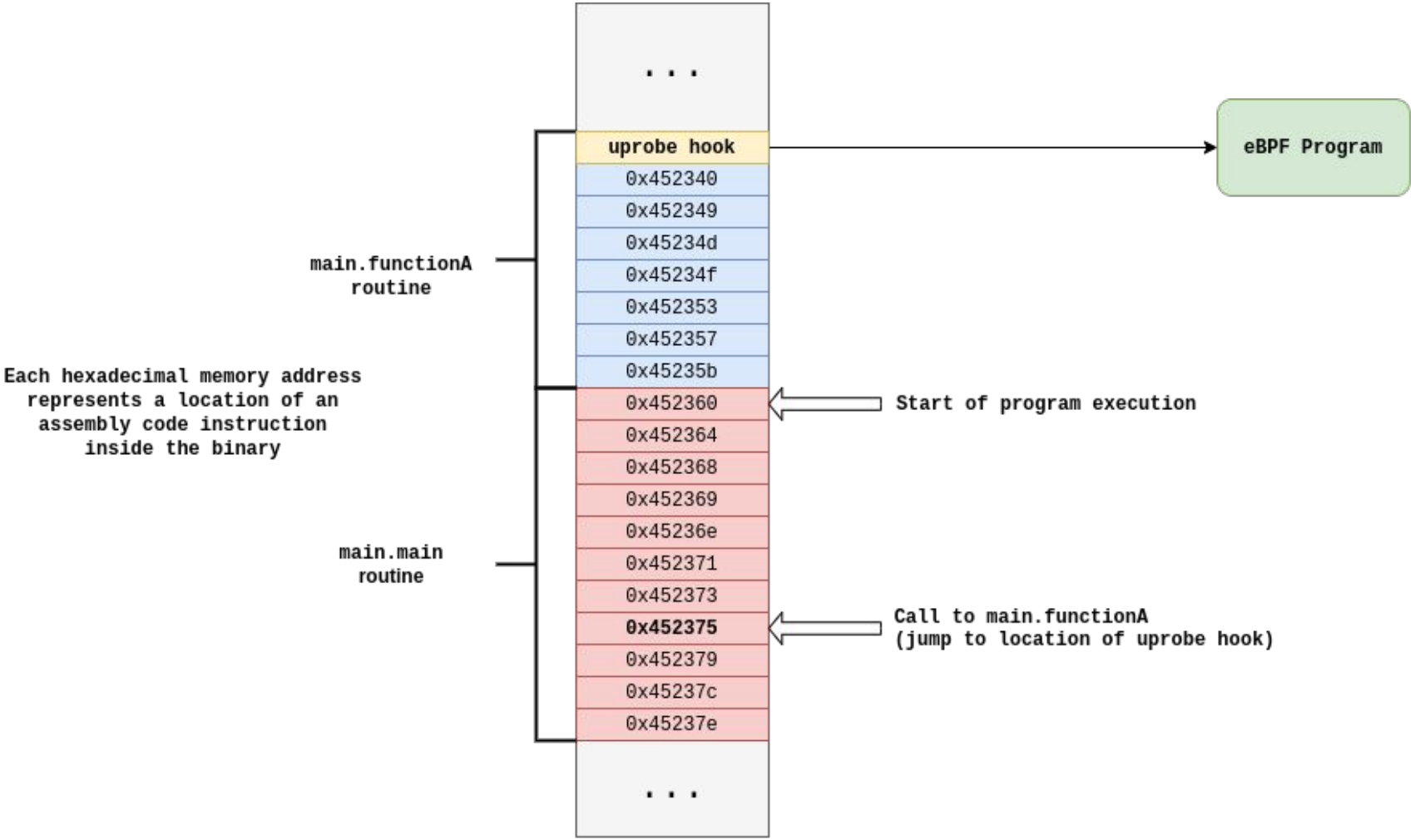
Let's start with some Go code

Slides/code: grant.pizza/talks/gophercon2020


1 package main
2
3 //go:noinline
4 func functionA() {}
5
6 func main() {
7 functionA()
8 }

```
SYMBOL main.functionA
0x4586b0:       ret


SYMBOL main.main
0x4586c0:       mov             rcx, qword ptr fs:[0xfffffffffffffff8]
0x4586c9:       cmp             rsp, qword ptr [rcx + 0x10]
0x4586cd:       jbe             0x4586e9
0x4586cf:       sub             rsp, 8
0x4586d3:       mov             qword ptr [rsp], rbp
0x4586d7:       lea             rbp, [rsp]
0x4586db:       call            0x4586b0                        # main.functionA
0x4586e0:       mov             rbp, qword ptr [rsp]
0x4586e4:       add             rsp, 8
0x4586e8:       ret
0x4586e9:       call            0x451eb0                        # runtime.morestack_noctxt
0x4586ee:       jmp             0x4586c0
```

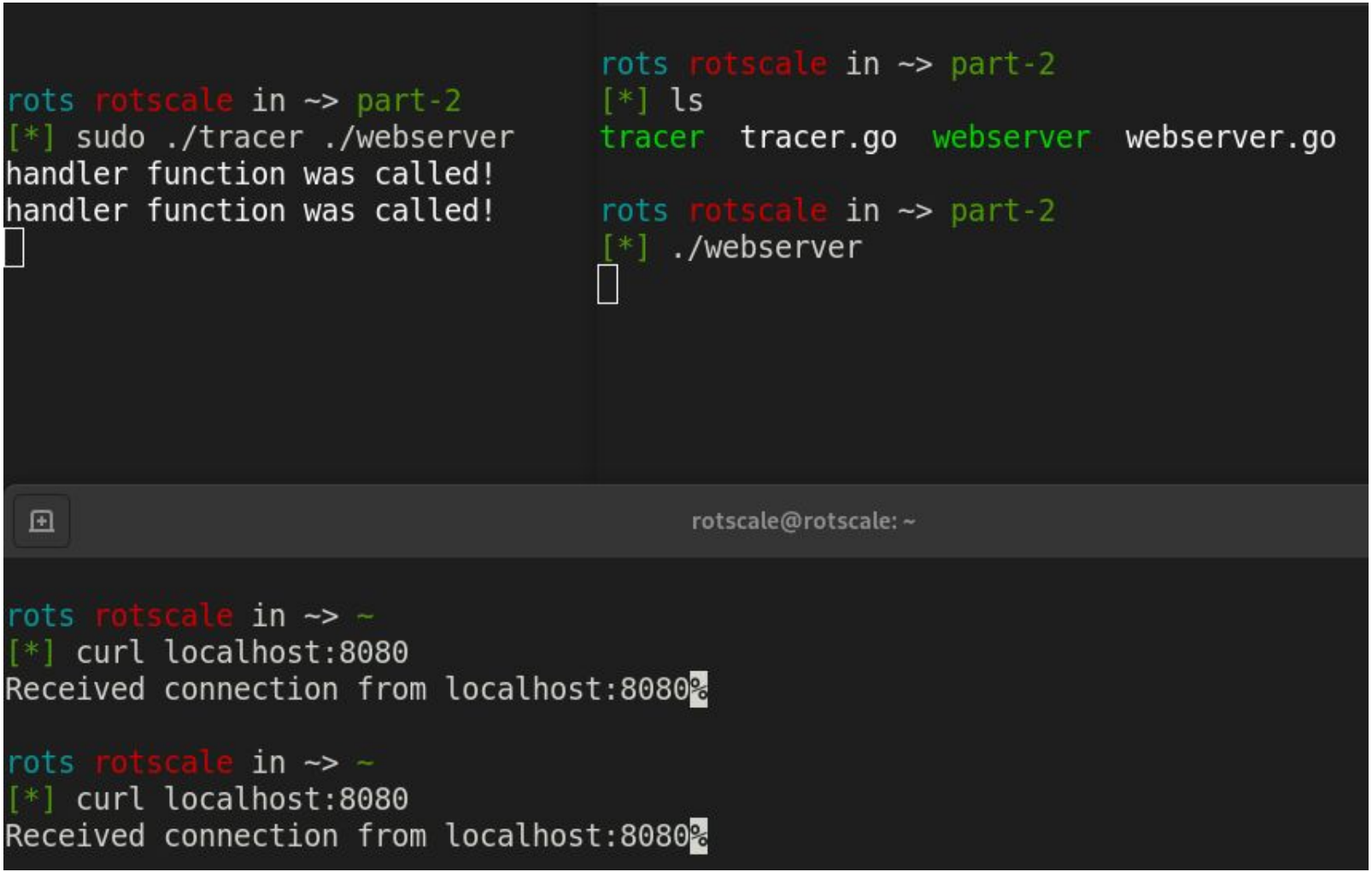

SYMBOL main.functionA
0x4586b0: ret
SYMBOL main.main
0x4586c0: mov rcx, qword ptr fs:[0xfffffffffffffff8]
0x4586c9: cmp rsp, qword ptr [rcx + 0x10]
0x4586cd: jbe 0x4586e9
0x4586cf: sub rsp, 8
0x4586d3: mov qword ptr [rsp], rbp
0x4586d7: lea rbp, [rsp]
0x4586db: call 0x4586b0 # main.functionA
0x4586e0: mov rbp, qword ptr [rsp]
0x4586e4: add rsp, 8
0x4586e8: ret
0x4586e9: call 0x451eb0 # runtime.morestack_noctxt
0x4586ee: jmp 0x4586c0


. . .
uprobe hook
0x452340
0x452349
0x45234d
0x45234f
0x452353
0x452357
0x45235b
0x452360
0x452364
0x452368
0x452369
0x45236e
0x452371
0x452373
0x452375
0x452379
0x45237c
0x45237e
. . .
eBPF Program
main.functionA routine
main.main routine
Each hexadecimal memory address represents a location of an assembly code instruction inside the binary
Start of program execution
Call to main.functionA (jump to location of uprobe hook)

```go
package main

import (
    "fmt"
    "log"
    "net/http"
)

//go:noinline
func handlerFunction(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Received connection from %s", r.Host)
}

func main() {
    http.HandleFunc("/", handlerFunction)
    log.Fatal(http.ListenAndServe(":8080", nil))
}
```

```
const eBPFProgram = `
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

BPF_PERF_OUTPUT(events);

int function_was_called(struct pt_regs *ctx) {
    char message[] = "handler function was called";
    events.perf_submit(ctx, &message, sizeof(message));
    return 0;
}
`
```

```
func main() {

    // BCC has a concept of 'module'. It's just a datastructure, not at all to be confused with kernel module
    bpfModule := bcc.NewModule(eBPFProgram, []string{})

    // Here we create a uprobe and associate it with our eBPF program 'function_was_called'
    uprobeFd, err := bpfModule.LoadUprobe("function_was_called")
    if err != nil {
        log.Fatal(err)
    }

    // Next we attach that uprobe to the symbol 'main.handlerFunction' in our go binary(path passed at command line).
    err = bpfModule.AttachUprobe(os.Args[1], "main.handlerFunction", uprobeFd, -1)
    if err != nil {
        log.Fatal(err)
    }

    // The code below is to set up the perf map on the userland side
    // This is to get output. There are other ways of doing this (bpf_trace_printk)
    // We're setting it up so we just get data from a channel of byte slices
    table := bcc.NewTable(bpfModule.TableId("events"), bpfModule)
    channel := make(chan []byte)
    perfMap, err := bcc.InitPerfMap(table, channel, nil)
    if err != nil {
        log.Fatal(err)
    }

    perfMap.Start()
    defer perfMap.Stop()

    // Start a go routine for reading from it
    go func() {
        for {
            value := <-channel
            fmt.Println(string(value))
        }
    }()

    // Listen for ctrl-c
    c := make(chan os.Signal, 1)
    signal.Notify(c, os.Interrupt)

    // Block
    <-c
}
```


rots rotscale in ~> part-2
[*] sudo ./tracer ./webserver
handler function was called!
handler function was called!
rots rotscale in ~> part-2
[*] ls
tracer tracer.go webserver webserver.go
rots rotscale in ~> part-2
[*] ./webserver
rotscale@rotscale: ~
rots rotscale in ~> ~
[*] curl localhost:8080
Received connection from localhost:8080%
rots rotscale in ~> ~
[*] curl localhost:8080
Received connection from localhost:8080%

Why this is useful

- Logging and debugging of programs that are already compiled, deployed, or running.
- Sometimes you don’t control the source code.

eBPF is a lot more powerful than just logging

eBPF programs have access to registers of the program they're tracing.

That includes a reference to the stack (*sp*).

```
struct pt_regs {
    unsigned long r15;
    unsigned long r14;
    unsigned long r13;
    unsigned long r12;
    unsigned long bp;
    unsigned long bx;
    unsigned long r11;
    unsigned long r10;
    unsigned long r9;
    unsigned long r8;
    unsigned long ax;
    unsigned long cx;
    unsigned long dx;
    unsigned long si;
    unsigned long di;
    unsigned long orig_ax;
    unsigned long ip;
    unsigned long cs;
    unsigned long flags;
    unsigned long sp;
    unsigned long ss;
};
```

```
package main

import (
    "log"
    "os"
    "os/signal"

    "github.com/iovisor/gobpf/bcc"
)

const eBPFProgram = `
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

int function_was_called(struct pt_regs *ctx) {

    struct task_struct *task;
    task = (struct task_struct *)bpf_get_current_task();
    u32 pid = task->tgid;
    u32 ppid = task->real_parent->tgid;

    bpf_trace_printk("main.handlerFunction() was called. PID: %d, PPID: %d\n", pid, ppid);

    return 0;
}
`
```

Tracing Go function arguments

- When switching contexts between different functions, the Go runtime will place arguments at offsets from the top of the stack based on their size and order in the list of arguments.
- Other languages, like C, put them in registers.

```
 1  package main
 2
 3  //go:noinline
 4  func functionB(x int, y int8) {
 5      print(x)
 6  }
 7
 8  func main() {
 9      functionB(42, 3)
10  }
11
```

```
SYMBOL main.functionB
0x4586b0:       mov             rcx, qword ptr fs:[0xfffffffffffffff8]
0x4586b9:       cmp             rsp, qword ptr [rcx + 0x10]
0x4586bd:       jbe             0x458701
0x4586bf:       sub             rsp, 0x10
0x4586c3:       mov             qword ptr [rsp + 8], rbp
0x4586c8:       lea             rbp, [rsp + 8]
0x4586cd:       call            0x42b2d0                        # runtime.printlock
0x4586d2:       mov             rax, qword ptr [rsp + 0x18]
0x4586d7:       mov             qword ptr [rsp], rax
0x4586db:       call            0x42ba50                        # runtime.printint
0x4586e0:       movzx           eax, byte ptr [rsp + 0x20]
0x4586e5:       movsx           rax, al
0x4586e9:       mov             qword ptr [rsp], rax
0x4586ed:       call            0x42ba50                        # runtime.printint
0x4586f2:       call            0x42b350                        # runtime.printunlock
0x4586f7:       mov             rbp, qword ptr [rsp + 8]
0x4586fc:       add             rsp, 0x10
0x458700:       ret
0x458701:       call            0x451eb0                        # runtime.morestack_noctxt
0x458706:       jmp             0x4586b0


SYMBOL main.main
0x458710:       mov             rcx, qword ptr fs:[0xfffffffffffffff8]
0x458719:       cmp             rsp, qword ptr [rcx + 0x10]
0x45871d:       jbe             0x458749
0x45871f:       sub             rsp, 0x18
0x458723:       mov             qword ptr [rsp + 0x10], rbp
0x458728:       lea             rbp, [rsp + 0x10]
0x45872d:       mov             qword ptr [rsp], 0x2a
0x458735:       mov             byte ptr [rsp + 8], 3
0x45873a:       call            0x4586b0                        # main.functionB
0x45873f:       mov             rbp, qword ptr [rsp + 0x10]
0x458744:       add             rsp, 0x18
0x458748:       ret
0x458749:       call            0x451eb0                        # runtime.morestack_noctxt
0x45874e:       jmp             0x458710
```

```
SYMBOL main.functionB
0x4586b0:       mov             rcx, qword ptr fs:[0xfffffffffffffff8]
0x4586b9:       cmp             rsp, qword ptr [rcx + 0x10]
0x4586bd:       jbe             0x458701
0x4586bf:       sub             rsp, 0x10
0x4586c3:       mov             qword ptr [rsp + 8], rbp
0x4586c8:       lea             rbp, [rsp + 8]
0x4586cd:       call            0x42b2d0                        # runtime.printlock
0x4586d2:       mov             rax, qword ptr [rsp + 0x18]
0x4586d7:       mov             qword ptr [rsp], rax
0x4586db:       call            0x42ba50                        # runtime.printint
0x4586e0:       movzx           eax, byte ptr [rsp + 0x20]
0x4586e5:       movsx           rax, al
0x4586e9:       mov             qword ptr [rsp], rax
0x4586ed:       call            0x42ba50                        # runtime.printint
0x4586f2:       call            0x42b350                        # runtime.printunlock
0x4586f7:       mov             rbp, qword ptr [rsp + 8]
0x4586fc:       add             rsp, 0x10
0x458700:       ret
0x458701:       call            0x451eb0                        # runtime.morestack_noctxt
0x458706:       jmp             0x4586b0


SYMBOL main.main
0x458710:       mov             rcx, qword ptr fs:[0xfffffffffffffff8]
0x458719:       cmp             rsp, qword ptr [rcx + 0x10]
0x45871d:       jbe             0x458749
0x45871f:       sub             rsp, 0x18
0x458723:       mov             qword ptr [rsp + 0x10], rbp
0x458728:       lea             rbp, [rsp + 0x10]
0x45872d:       mov             qword ptr [rsp], 0x2a
0x458735:       mov             byte ptr [rsp + 8], 3
0x45873a:       call            0x4586b0                        # main.functionB
0x45873f:       mov             rbp, qword ptr [rsp + 0x10]
0x458744:       add             rsp, 0x18
0x458748:       ret
0x458749:       call            0x451eb0                        # runtime.morestack_noctxt
0x45874e:       jmp             0x458710
```

How to trace Go function arguments

- Access the top of the stack using the **`sp`** field of **`pt_regs`**
- Calculate the arguments offset off the stack
- Read those arguments off the stack using **`bpf_probe_read`**
- Send them up to userspace via the perf buffer
- Parse them in Go using the **`encoding/binary`** package

Calculating stack offsets

- Arguments start 8 bytes off the stack (first 8 is for base pointer)
- Parameters are padded on the stack based on the largest data type amongst them. (limit: 8)
- ‘Window’ size based on largest data type.
- Parameters are put on the stack based on if they fit in the current ‘window’.
- If they don’t, the stack is padded until the start of the next ‘window’.
- Parameter ordering (kinda) affects memory efficiency!

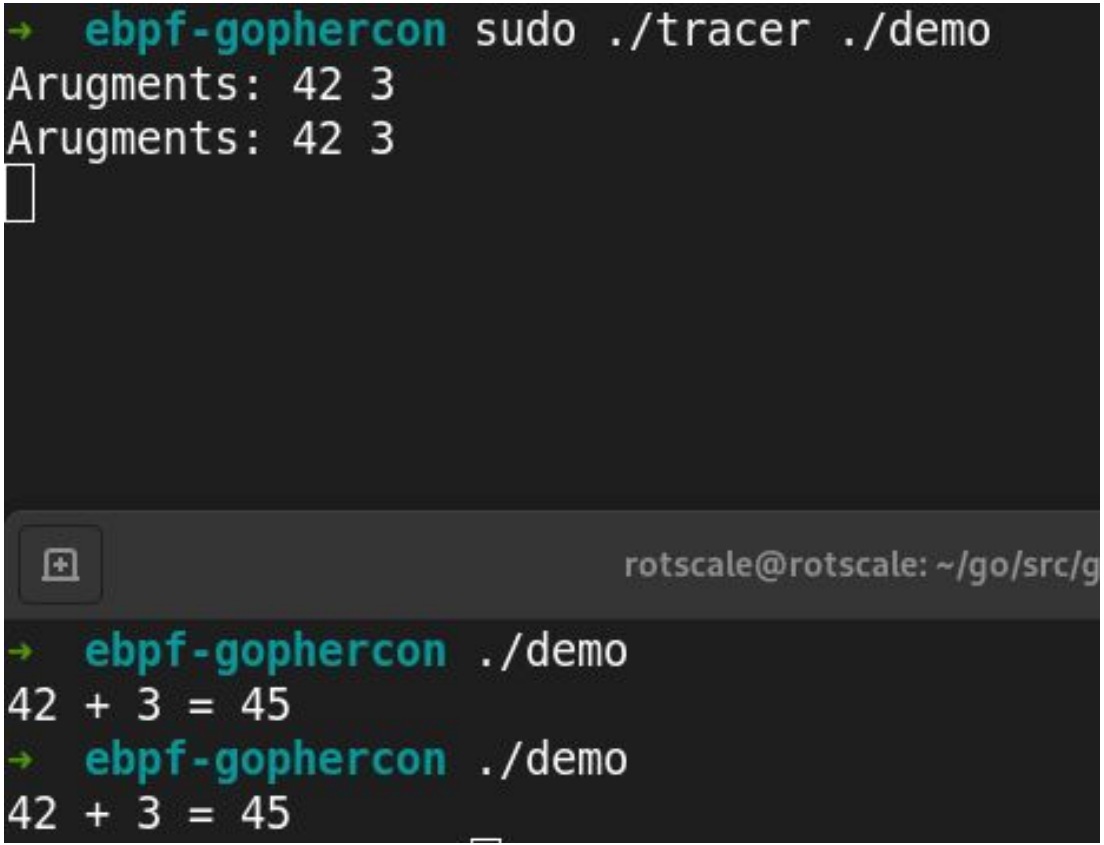
Datatypes	Size (in bytes)
int8, uint8	1
int16, uint16	2
int32, uint32	4
int, uint, int64, uint64	8
float32	4
float64	8
bool	1
byte	1
rune	4
string	16
pointers	8
structs	8
interfaces	16

- `int` is 8 bytes, `int8` is 1 byte
- The first parameter starts at `sp+8` and takes up 8 bytes
- The second parameter starts at `sp+15` and takes up 1 byte followed by 7 bytes of padding

```
1   package main
2
3   import "fmt"
4
5   //go:noinline
6   func addTwoNumbers(x int, y int8) {
7       fmt.Printf("%d + %d = %d\n", x, y, x+int(y))
8   }
9
10  func main() {
11      addTwoNumbers(42, 3)
12  }
13
```

```
const eBPFProgram = `
#include <uapi/linux/ptrace.h>

BPF_PERF_OUTPUT(events);

inline int print_stack(struct pt_regs *ctx) {
    void* stackAddr = (void*)ctx->sp;

    long argument1;
    bpf_probe_read(&argument1, sizeof(argument1), stackAddr+8);
    events.perf_submit(ctx, &argument1, sizeof(argument1));

    char argument2;
    bpf_probe_read(&argument2, sizeof(argument2), stackAddr+16);
    events.perf_submit(ctx, &argument2, sizeof(argument2));

    return 0;
}
`
```

```
// Start a go routine for reading from it
go func() {

    for {
        firstBytes := <-channel
        first := binary.LittleEndian.Uint64(firstBytes)
        secondBytes := <-channel
        second := binary.LittleEndian.Uint32(secondBytes)
        fmt.Println(first, second)
    }
}()
```


➜ ebpf-gophercon sudo ./tracer ./demo
Arugments: 42 3
Arugments: 42 3
rotscale@rotscale: ~/go/src/g
➜ ebpf-gophercon ./demo
42 + 3 = 45
➜ ebpf-gophercon ./demo
42 + 3 = 45

```
13  const eBPFProgram = `
14  #include <uapi/linux/ptrace.h>
15  #include <linux/sched.h>
16
17  BPF_PERF_OUTPUT(events);
18
19  int function_was_called(struct pt_regs *ctx) {
20      void* stackAddr = (void*)ctx->sp;
21
22      long newArgument1 = 7;
23      bpf_probe_write_user(stackAddr+8, &newArgument1, sizeof(newArgument1));
24
25      return 0;
26  }
27  `
```

Weaver

Weaver is a CLI tool that allows you to trace Go programs in order to inspect what values are passed to specified functions. It leverages eBPF attached to uprobes.

Quick Start

There are two modes of operation, one that uses a 'functions file', and one that extracts a symbol table from a passed binary and filters by Go packages. More information on functionality in docs.

So many possibilities

Great place to learn more about eBPF is ebpf.io

Thank you so much!

grantseltzer@gmail.com

grant.pizza

github.com/grantseltzer