Redactable Logs for CockroachDB

In v20.2 and perhaps v20.1/v19.2

Raphael 'kena' Poss - 2020-06-08

Cockroach LABS

Summary

Redactable Logs for CockroachDB

In a nutshell

Log files on-disk retain all details, unsafe data can be **automatically** stripped out

How: **redaction markers** around unsafe bits — for example:

```
[n1,consistencyChecker,s2,r4/1:‹/System{/tsd-tse}›] triggering stats
recomputation to resolve delta of ‹{ContainsEstimates:1438  ...}›
```

Then: `debug zip --redact-logs` — for example:

```
[n1,consistencyChecker,s2,r4/1:‹×›] triggering stats recomputation to resolve
delta of ‹×›
```

Redactable Logs for CockroachDB

Why?

- **Compliance**:
  - JPMC and others bank-like customers — simply can't share, financial regulations
  - **GDPR** and similar forces CRL to become “data processor” to receive PII and other confidential data from any support customers — this is costly and legal minefield
  - Creates a confidentiality barrier for CockroachCloud
- **Customers have often asked us** for this
- Might enable **more sales** of our support services if customers don't feel we spy

Redactable Logs for CockroachDB

Key terms and concepts

Data is **safe** if it is ***guaranteed*** / proven ***not to contain information*** that a customer *may* not want to share with us: PII; confidential information; legally protected information; etc.

Data is **unsafe otherwise**.

- Visibly contains PII, confidential etc — obviously unsafe
- *Visibly* does not contain PII, confidential etc, but not *proven* not to, is also unsafe

This is a conservative approach: we consider anything unsafe until we have very good reasons not to; *everything gets redacted* except for those bits which we know are safe

How this impacts customers and CRL

What changes for whom — Users

New server option: **--redactable-logs**

Defaults to **true** (enabled) in v20.2

Defaults to **false** in v20.1 and v19.2 (if feature gets backported — TBD)

New client option for `debug zip`: **--redact-logs**

Might default to **true** in v20.2 (TBD)

Messaging: *redaction occurs server-side; no sensitive data travels over the network*

What changes for whom — Technical Support

See previous slide: **--redactable-logs** / **--redact-logs**

NB: `cockroach debug zip --redact-logs` can redact (very conservatively)

even when `--redactable-logs` was not enabled server-side.

However *this makes logs nearly unusable* — **recommend --redactable-logs=true always**

Company policy should transition over time to **request *redacted* logs first**

This sustains customer trust and minimizes legal exposure of CRL

What changes for whom — Cockroach Cloud

New server option: **--redactable-logs**

Will aim to **enable always** in CockroachCloud clusters

When escalating an issue from CC to Tech Support / Engineering:

- Provide redacted logs first
- Only after additional request/escalation, provide full logs
    - *Process: has the person who wants full logs sufficient credentials to access CC customer's data?*

What changes for whom — Documentation

We need to document the command-line options (obvs...) + recommend --redactable-logs

We need to ***document the logging format***

- Allows users to audit the correctness of our redaction algorithms
- **This builds trust and confidence**
- **Creates value** by enabling 3rd party monitoring that is confidentiality-aware

Update our **Responsible disclosure policy**

Redaction failures are to be reported as security vulnerabilities

What changes for whom — Engineering

- *I am receiving a redacted log from support / a test failure — now what?*
    - The assumption is that the remaining data is sufficient for you to do your job
    - If it is not, the priority should be to enhance the logging ahead of negotiating for unredacted logs (take urgency of situation into account)
- *How do I make my logging code redactable? How do I enhance it towards this?*
    - See slides at end with examples

- *I found a bug which causes unsafe data to be preserved in redacted logs*
    - Treat this as security vulnerability and talk to #security / security@

The Plan™

How this will come to fruition

1. Design+impl infrastructure in CockroachDB `master` — done (RFC, PR)
2. Socialize the approach — *you are here!*
3. Enable in testing internally (June-August 2020)
    - Test logs show redacted logs first, extra work needed to see all
    - This nudges all engineers to improve logging for redactability
4. Concurrently with #3, iterate on API and log calls based on experience
5. Concurrently with #3, impl redaction for more pieces of `debug zip`, not just logs
6. Set up external docs for users + workflows / explanations for Technical Support
7. Feature + processes ready for v20.2.   *(Currently discussing feasibility of backport.)*

Questions?

- Ask me directly
- Discuss implementation on #kv / #support

Engineering: Code Updates

Technical Approach

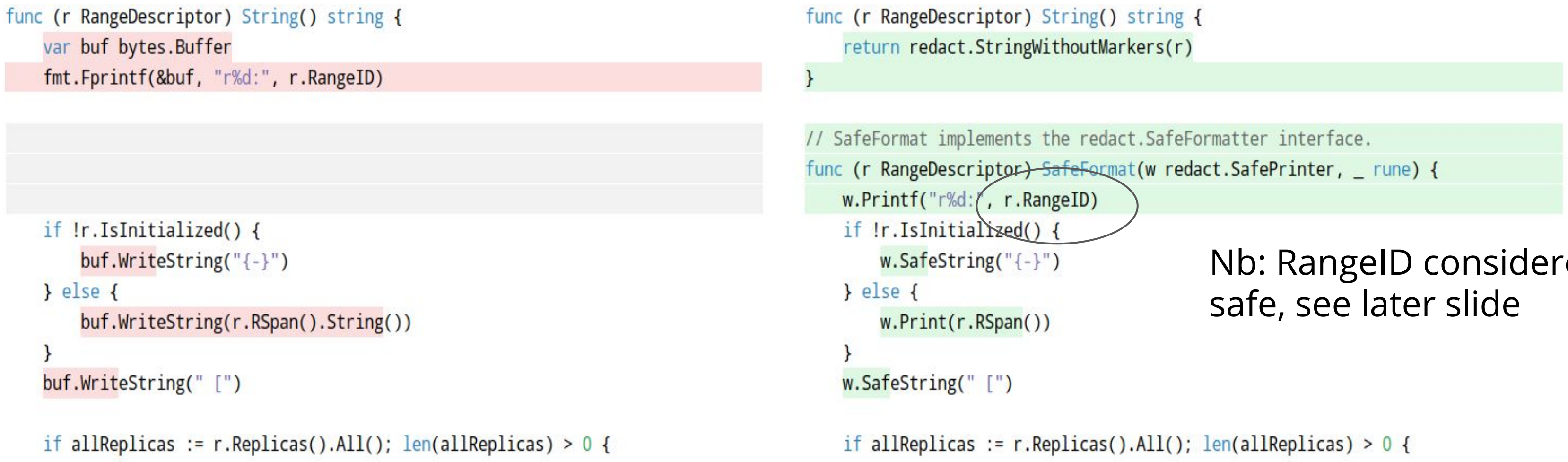
- **Log API calls do not change** — mostly:
  - The *format* string of Infof(..) calls is considered always safe
  - Therefore, we lint it to mandate it be a constant
  - E.g. `log.Infof("my string " + myVar)` is now invalid

    Use `log.Infof("my string %s", myVar)` instead
- Each value to be logged can decide to "make itself redactable" or not
  - Via **SafeFormat()** method (main), for leaf/simple types only **SafeValue()**
  - There's also a global registry or pre-defined always-safe types, eg time.Duration
- log.Safe(...) still exists but is now being demoted (evt deprecated)
-

Common Case: String() to SafeFormat()

(Examples from https://github.com/cockroachdb/cockroach/pull/48051)

E.g. roachpb/metadata.go:



```go
func (r RangeDescriptor) String() string {
	var buf bytes.Buffer
	fmt.Fprintf(&buf, "r%d:", r.RangeID)




	if !r.IsInitialized() {
		buf.WriteString("{-}")
	} else {
		buf.WriteString(r.RSpan().String())
	}
	buf.WriteString(" [")

	if allReplicas := r.Replicas().All(); len(allReplicas) > 0 {
```

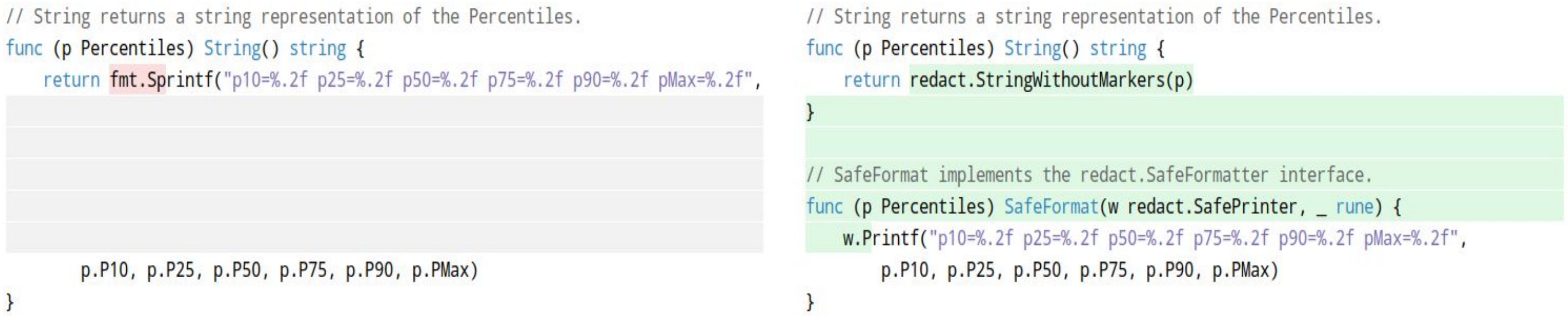
```go
func (r RangeDescriptor) String() string {
	return redact.StringWithoutMarkers(r)
}

// SafeFormat implements the redact.SafeFormatter interface.
func (r RangeDescriptor) SafeFormat(w redact.SafePrinter, _ rune) {
	w.Printf("r%d:", r.RangeID)
	if !r.IsInitialized() {
		w.SafeString("{-}")
	} else {
		w.Print(r.RSpan())
	}
	w.SafeString(" [")

	if allReplicas := r.Replicas().All(); len(allReplicas) > 0 {
```

Nb: RangeID considered safe, see later slide

Common Case: String() to SafeFormat()

(Examples from https://github.com/cockroachdb/cockroach/pull/48051)

Simple numeric values (bool, ints, floats) are always considered safe:



```go
// String returns a string representation of the Percentiles.
func (p Percentiles) String() string {
    return fmt.Sprintf("p10=%.2f p25=%.2f p50=%.2f p75=%.2f p90=%.2f pMax=%.2f",
        p.P10, p.P25, p.P50, p.P75, p.P90, p.PMax)
}
```

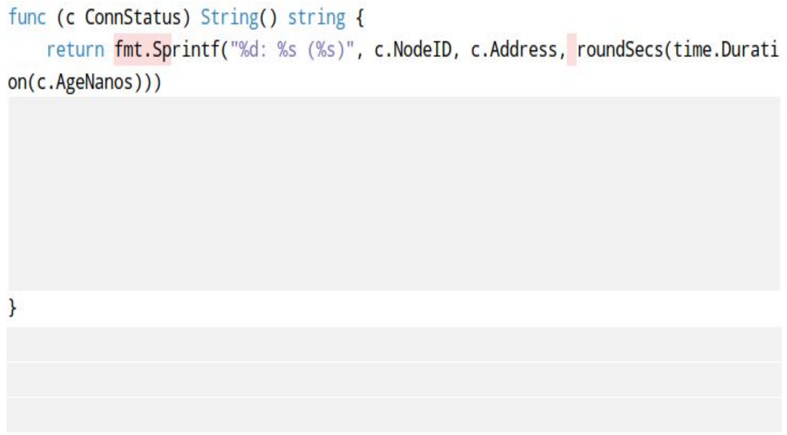
```go
// String returns a string representation of the Percentiles.
func (p Percentiles) String() string {
    return redact.StringWithoutMarkers(p)
}

// SafeFormat implements the redact.SafeFormatter interface.
func (p Percentiles) SafeFormat(w redact.SafePrinter, _ rune) {
    w.Printf("p10=%.2f p25=%.2f p50=%.2f p75=%.2f p90=%.2f pMax=%.2f",
        p.P10, p.P25, p.P50, p.P75, p.P90, p.PMax)
}
```

Common Case: String() to SafeFormat()

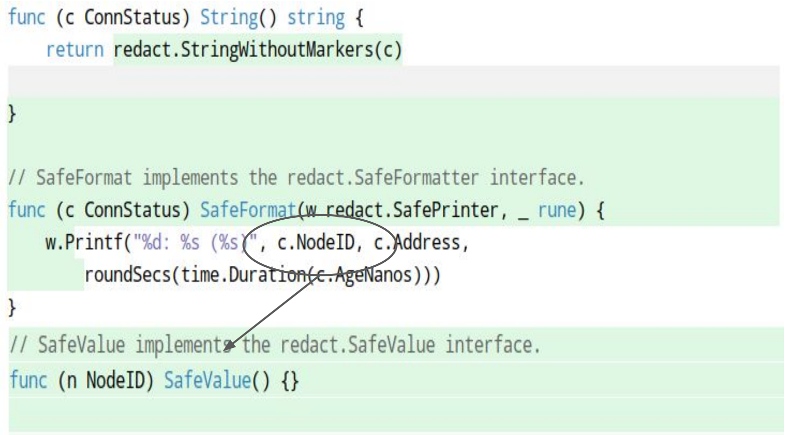
(Examples from https://github.com/cockroachdb/cockroach/pull/48051)

SafeFormat() *recursively delegates* the creation of redactable output

Recursion terminates at either unsafe data, or always-safe leaf/simple value



```go
func (c ConnStatus) String() string {
    return fmt.Sprintf("%d: %s (%s)", c.NodeID, c.Address, roundSecs(time.Durati
on(c.AgeNanos)))
}
```



```go
func (c ConnStatus) String() string {
    return redact.StringWithoutMarkers(c)

}

// SafeFormat implements the redact.SafeFormatter interface.
func (c ConnStatus) SafeFormat(w redact.SafePrinter, _ rune) {
    w.Printf("%d: %s (%s)", c.NodeID, c.Address,
        roundSecs(time.Duration(c.AgeNanos)))
}
// SafeValue implements the redact.SafeValue interface.
func (n NodeID) SafeValue() {}
```

Use SafeValue with caution
— see notes at end

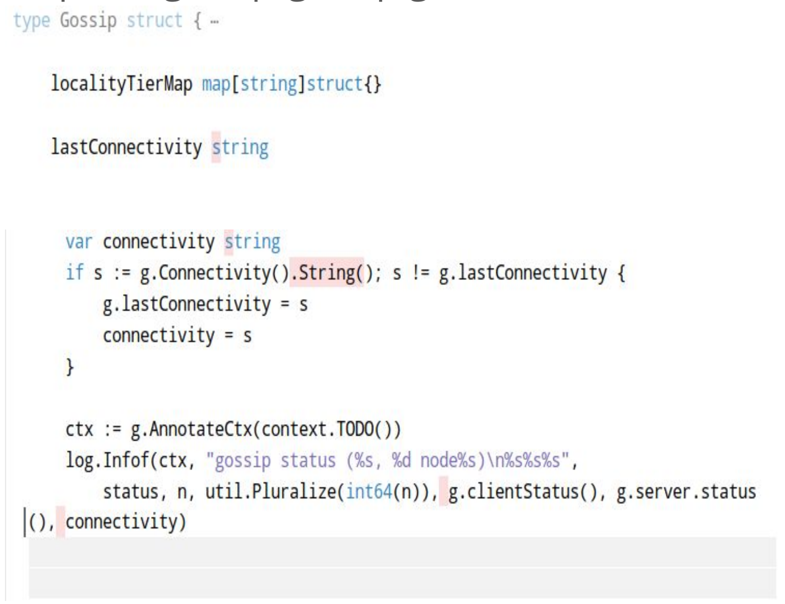
Cockroach LABS

@cockroachdb

Store Redactable in memory, log it later

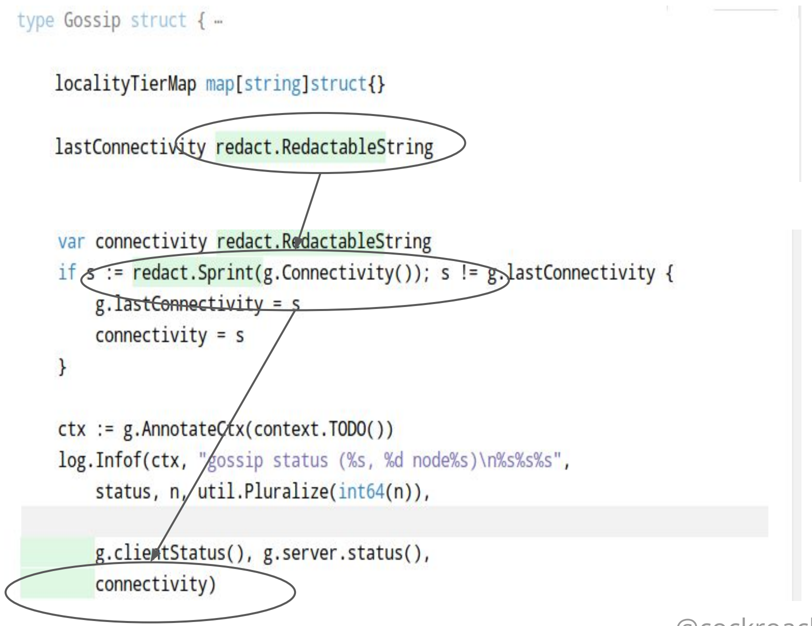
(Examples from https://github.com/cockroachdb/cockroach/pull/48051)

Example in gossip/gossip.go



```
type Gossip struct { …

    localityTierMap map[string]struct{}

    lastConnectivity string
```

```
    var connectivity string
    if s := g.Connectivity().String(); s != g.lastConnectivity {
        g.lastConnectivity = s
        connectivity = s
    }

    ctx := g.AnnotateCtx(context.TODO())
    log.Infof(ctx, "gossip status (%s, %d node%s)\n%s%s%s",
        status, n, util.Pluralize(int64(n)), g.clientStatus(), g.server.status
(), connectivity)
```



```
type Gossip struct { …

    localityTierMap map[string]struct{}

    lastConnectivity redact.RedactableString
```

```
    var connectivity redact.RedactableString
    if s := redact.Sprint(g.Connectivity()); s != g.lastConnectivity {
        g.lastConnectivity = s
        connectivity = s
    }

    ctx := g.AnnotateCtx(context.TODO())
    log.Infof(ctx, "gossip status (%s, %d node%s)\n%s%s%s",
        status, n, util.Pluralize(int64(n)),

        g.clientStatus(), g.server.status(),
        connectivity)
```

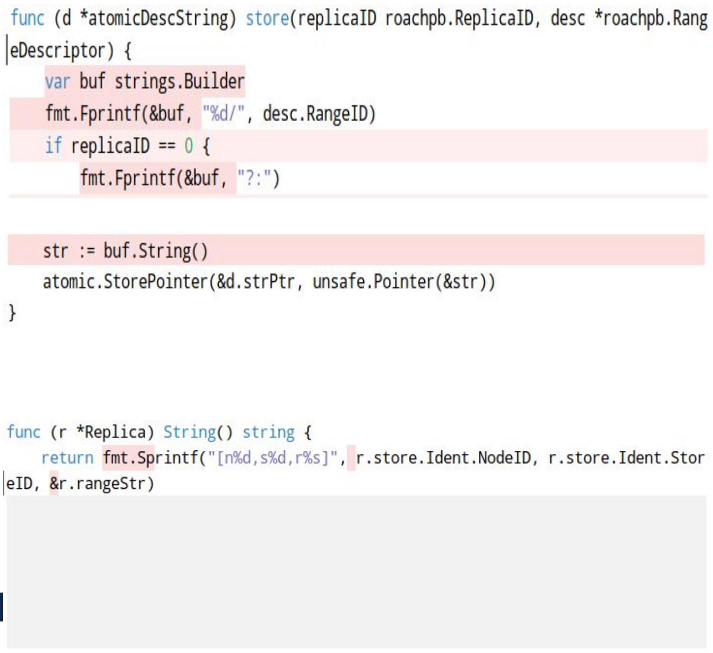
Cockroach LABS

@cockroachdb

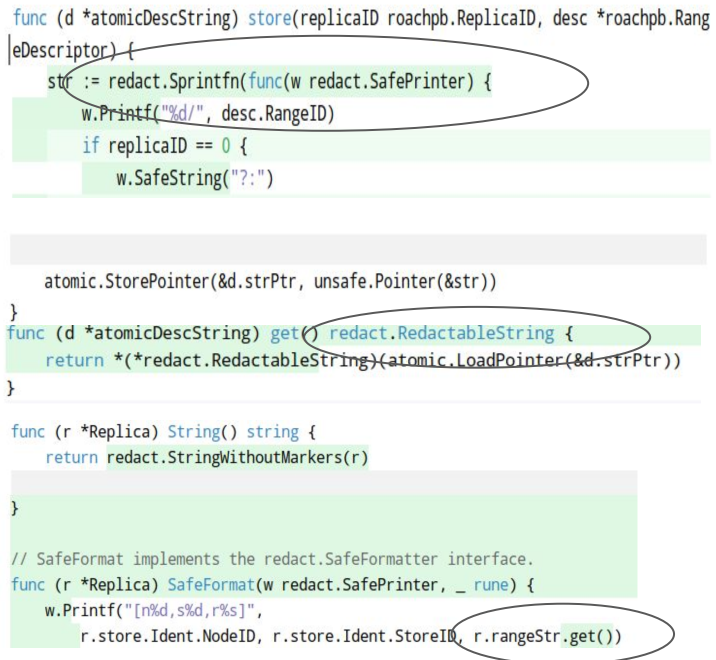
Store Redactable in memory, log it later

(Examples from https://github.com/cockroachdb/cockroach/pull/48051)

A more advanced example: the replica "range description string"



```
func (d *atomicDescString) store(replicaID roachpb.ReplicaID, desc *roachpb.Rang
eDescriptor) {
    var buf strings.Builder
    fmt.Fprintf(&buf, "%d/", desc.RangeID)
    if replicaID == 0 {
        fmt.Fprintf(&buf, "?:")

    str := buf.String()
    atomic.StorePointer(&d.strPtr, unsafe.Pointer(&str))
}


func (r *Replica) String() string {
    return fmt.Sprintf("[n%d,s%d,r%s]", r.store.Ident.NodeID, r.store.Ident.Stor
eID, &r.rangeStr)
```



```
func (d *atomicDescString) store(replicaID roachpb.ReplicaID, desc *roachpb.Rang
eDescriptor) {
    str := redact.Sprintfn(func(w redact.SafePrinter) {
        w.Printf("%d/", desc.RangeID)
        if replicaID == 0 {
            w.SafeString("?:")


    atomic.StorePointer(&d.strPtr, unsafe.Pointer(&str))
}
func (d *atomicDescString) get() redact.RedactableString {
    return *(*redact.RedactableString)(atomic.LoadPointer(&d.strPtr))
}

func (r *Replica) String() string {
    return redact.StringWithoutMarkers(r)

}

// SafeFormat implements the redact.SafeFormatter interface.
func (r *Replica) SafeFormat(w redact.SafePrinter, _ rune) {
    w.Printf("[n%d,s%d,r%s]",
        r.store.Ident.NodeID, r.store.Ident.StoreID, r.rangeStr.get())
```

Buffer a RedactableString incrementally

Before:

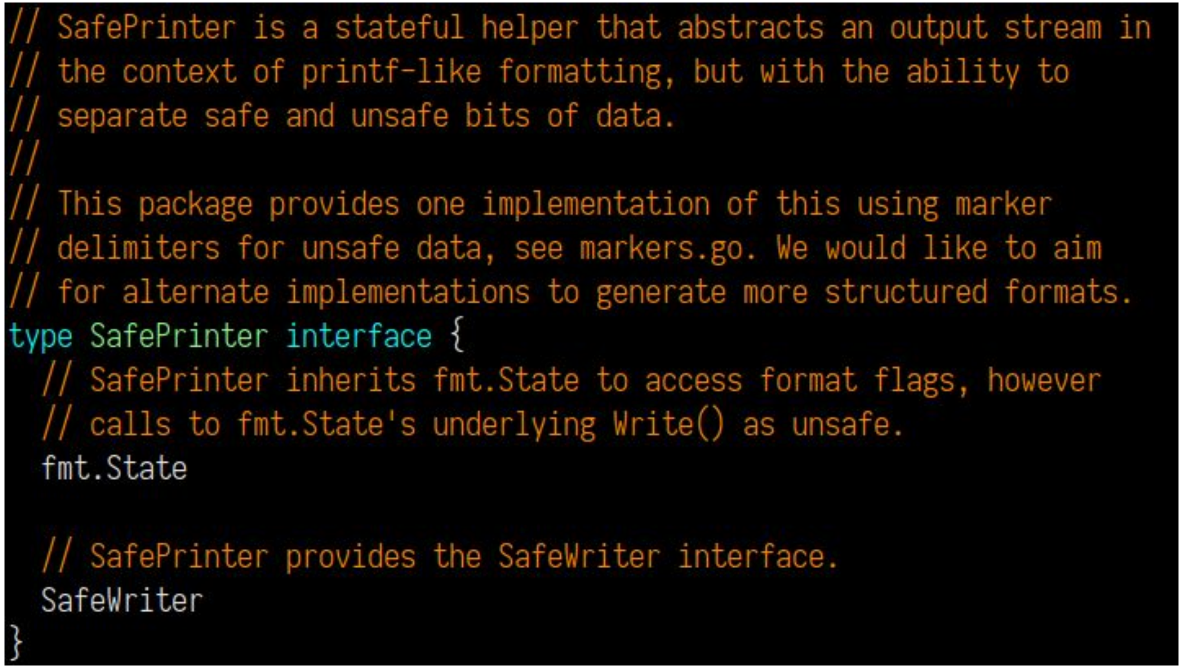
```
var buf strings.Builder
buf.WriteString("hello")
buf.WriteString("world")
fmt.Fprintf(&buf, "hello %s", "universe")
result := buf.String()
```

After:

```
var buf redact.StringBuilder
buf.SafeString("hello")
buf.UnsafeString("world")
buf.Printf("hello %s", "universe")
result := buf.RedactableString()

// NB: fmt.Fprintf(&buf) also works but
// considers everything printed as unsafe
```

What's in a SafePrinter?

The first arg to **SafeFormat(w redact.SafePrinter, verb rune)** methods



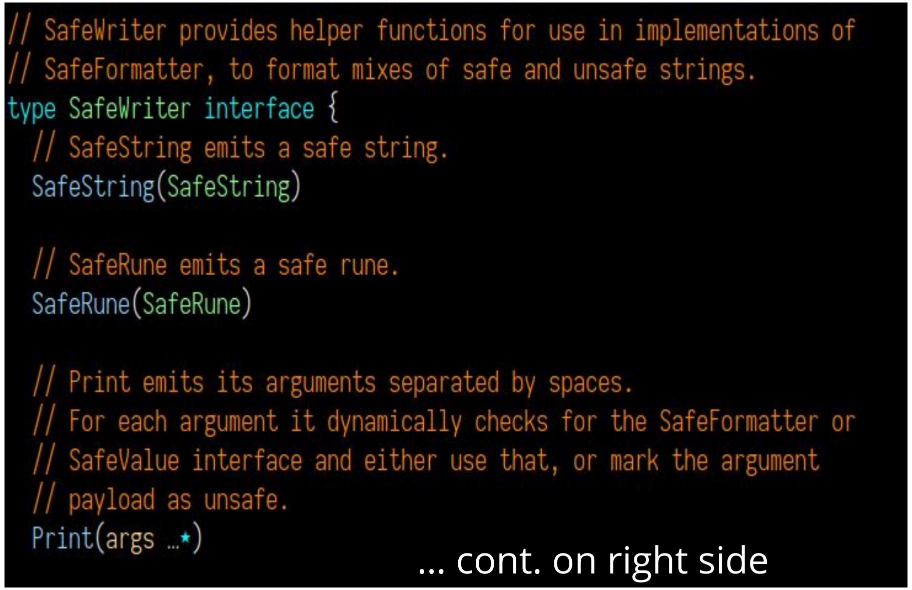
```
// SafePrinter is a stateful helper that abstracts an output stream in
// the context of printf-like formatting, but with the ability to
// separate safe and unsafe bits of data.
//
// This package provides one implementation of this using marker
// delimiters for unsafe data, see markers.go. We would like to aim
// for alternate implementations to generate more structured formats.
type SafePrinter interface {
  // SafePrinter inherits fmt.State to access format flags, however
  // calls to fmt.State's underlying Write() as unsafe.
  fmt.State

  // SafePrinter provides the SafeWriter interface.
  SafeWriter
}
```

Nb: fmt.State also implements io.Writer

Familiarity
With fmt.Formatter is advised

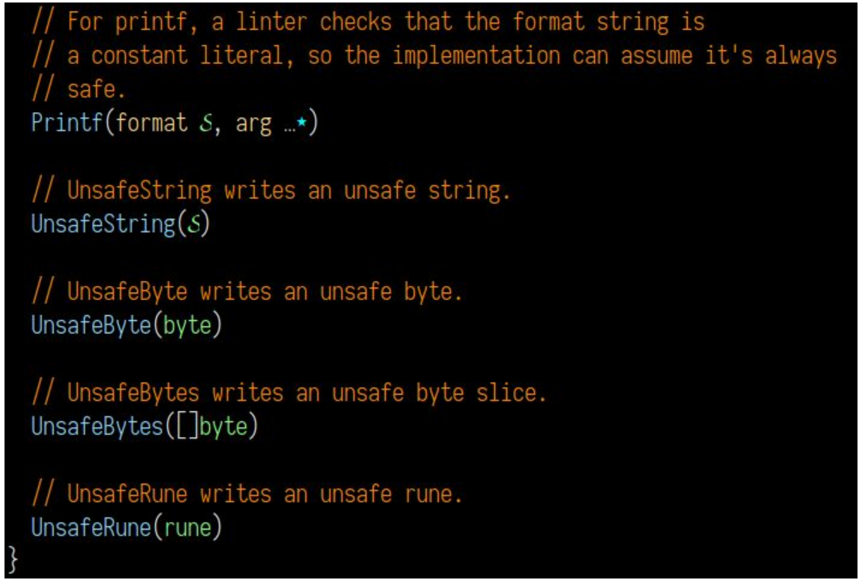
This also explains the "rune" 2nd arg

What's in a SafePrinter? (cont.)



```
// SafeWriter provides helper functions for use in implementations of
// SafeFormatter, to format mixes of safe and unsafe strings.
type SafeWriter interface {
  // SafeString emits a safe string.
  SafeString(SafeString)

  // SafeRune emits a safe rune.
  SafeRune(SafeRune)

  // Print emits its arguments separated by spaces.
  // For each argument it dynamically checks for the SafeFormatter or
  // SafeValue interface and either use that, or mark the argument
  // payload as unsafe.
  Print(args ...★)
```

... cont. on right side

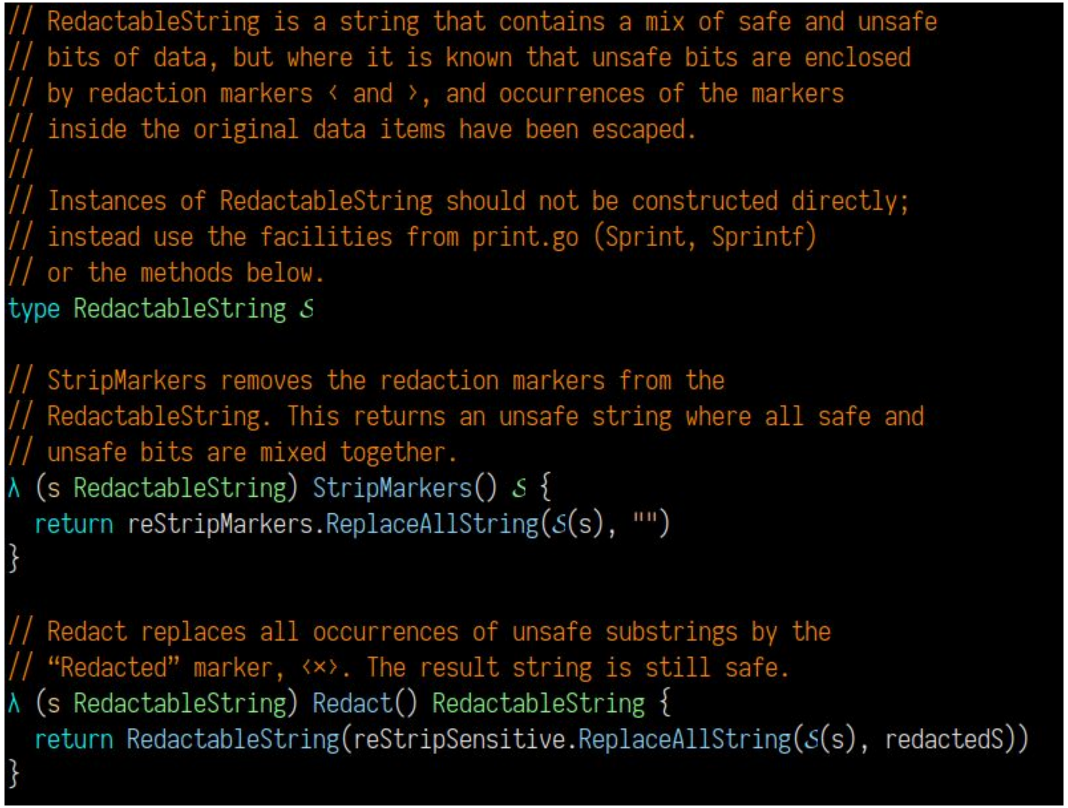


```
  // For printf, a linter checks that the format string is
  // a constant literal, so the implementation can assume it's always
  // safe.
  Printf(format 𝓢, arg ...★)

  // UnsafeString writes an unsafe string.
  UnsafeString(𝓢)

  // UnsafeByte writes an unsafe byte.
  UnsafeByte(byte)

  // UnsafeBytes writes an unsafe byte slice.
  UnsafeBytes([]byte)

  // UnsafeRune writes an unsafe rune.
  UnsafeRune(rune)
}
```

Cockroach LABS

@cockroachdb

What's a RedactableString?



```
// RedactableString is a string that contains a mix of safe and unsafe
// bits of data, but where it is known that unsafe bits are enclosed
// by redaction markers ‹ and ›, and occurrences of the markers
// inside the original data items have been escaped.
//
// Instances of RedactableString should not be constructed directly;
// instead use the facilities from print.go (Sprint, Sprintf)
// or the methods below.
type RedactableString 𝓈

// StripMarkers removes the redaction markers from the
// RedactableString. This returns an unsafe string where all safe and
// unsafe bits are mixed together.
λ (s RedactableString) StripMarkers() 𝓈 {
  return reStripMarkers.ReplaceAllString(𝓈(s), "")
}

// Redact replaces all occurrences of unsafe substrings by the
// “Redacted” marker, ‹×›. The result string is still safe.
λ (s RedactableString) Redact() RedactableString {
  return RedactableString(reStripSensitive.ReplaceAllString(𝓈(s), redactedS))
}
```

Cockroach LABS

@cockroachdb

In summary — when is data safe?

- If it's printed via **p.SafeString()** / **p.SafeRune()** from within a **SafeFormat**() method
- If it's enclosed in **log.Safe()** in a log call or SafeFormat method (deprecated)
- If it was in a p.Printf/log.XXf **format string** (as a constant, e.g. a literal)
- If it's not enclosed within redaction markers in **RedactableString** values
- If it is a value of a **registered always-safe type**
  - Static registry: all types implementing **SafeValue()**
  - Dynamic registry: non-aliased Go type **bool, int (incl int32 uint32 etc), float**, also **time.Duration, time.Time, hlc.Timestamp**
  - Reported in docs/generated/redact_safe.md, **extra scrutiny during reviews**

What's wrong with log.Safe() and SafeValue()

Consider: `log.Infof(ctx, "hello %s", log.Safe(myVar))`

Critical flaw: nothing prevents the definition of myVar from being changed, far from the log call, to start leaking unsafe information. There's no incentive/signal during reviews to care for this. **The same problem exists with SafeValue().**

Therefore we restrict SafeValue() to the most simple Go types. We'll also deprecate log.Safe(). Use SafeFormat() and RedactableString instead.

In summary — when is data unsafe?

General rule: **data is unsafe unless explicitly marked as safe** as per previous slides

In particular:

- Go "string" type always to be considered unsafe
  (who knows where a string comes from)
- String() methods always to be considered unsafe
  (too much risk of auto-call of a String() from 3rd party package)
- If you personally can't prove it's safe, consider it unsafe (*better be safe than sorry*)