24 Information security and recovery

AUTHOR
R. Heller

R. Heller · Date: 2026-07-30 · Last changed: 2026-11-09

At a glance

Guiding question: How does a team protect findings data, the model version and availability?
For: clinical IT, research software engineers, data stewards, and the clinical leads who decide whether a tool stays on; the core reading needs no programming. · **Prerequisites:** a reproducible R project from Chapter 4; monitoring and restart in Chapter 20; the data flow in Chapter 23.
In the course: W7 · Regulation, data protection and incident (control matrix and table-top exercise), W8 · Colloquium and decision · **Data:** synthetic (model file, logs and test cases generated in the chapter) · **Core reading time:** about 11 minutes

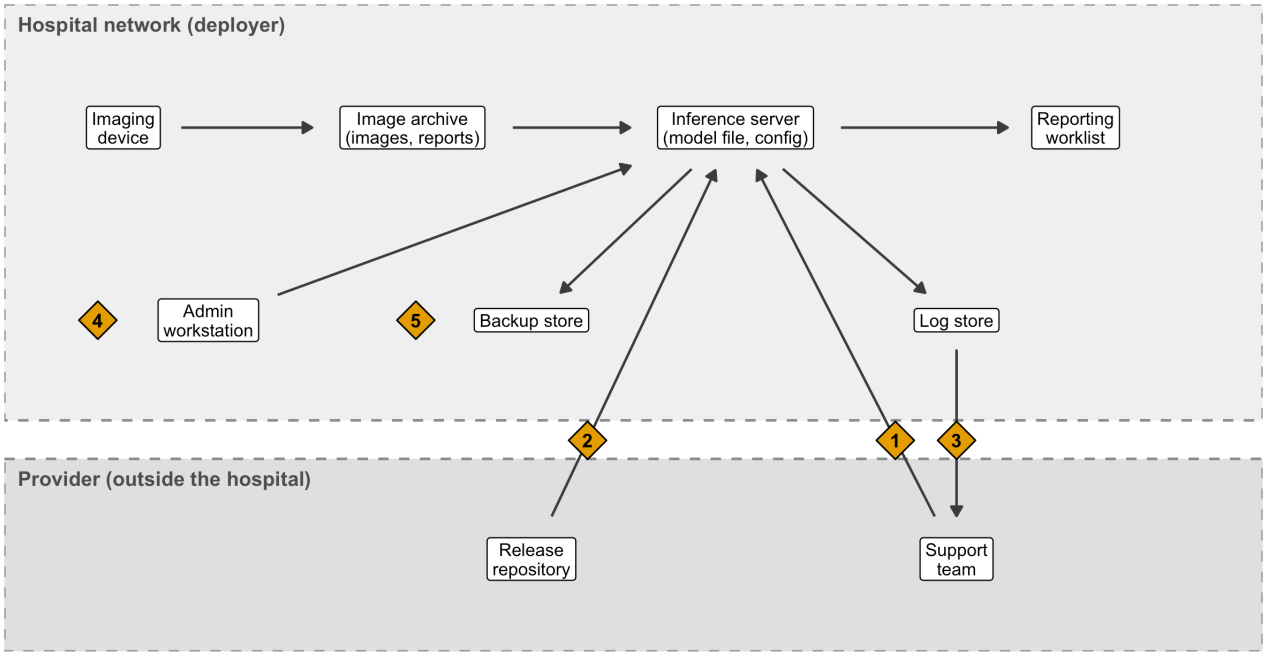
24.1 Case and decision

Monday, 07:40, in the fictitious emergency department of Chapter 9. The service provider of the fracture alert reports that one of its support logs contains part of a radiology report. Ten minutes later, the routine release check on the inference server shows that the installed model file does not have the checksum of the approved release. The consultant in charge has to decide whether the alert stays on. She switches it off. All radiographs now go through the regular reporting pathway, as on any day without the tool, so no patient waits longer for a report. The IT security officer blocks the provider’s remote access, secures the logs and starts a restore from a verified copy. The alert will come back only after a synthetic test case has passed and the system owner has signed. The night shift works without the second look; nobody works without a report.

24.2 Core reading

Core reading · recommended reading for W7 (table-top material) · about 11 minutes · ends before ‘Lab’

A threat model in four questions. What are we protecting (assets)? Where can it be attacked (attack surface)? Where does trust change (trust boundaries)? Who is responsible for each part (owners)? Figure 24.1 answers these questions for the fictional second-look service. Radiographs go from the imaging device to the image archive, the inference server reads them, applies the model and writes a marker back to the reporting worklist. The model arrives from the provider’s release repository; the provider’s support team can connect remotely; logs go to a log store; backups go to a separate store. Five points of the attack surface are numbered. Only one of them – the remote support access – is visible to clinicians, and the incident in the case used two of them.



Fictional system · orange diamonds = attack surface (1–5)

Figure 24.1: Threat model of the fictional second-look service. Boxes are components, arrows are data flows, shaded areas are trust boundaries. Numbered diamonds mark the attack surface: (1) remote support access, (2) release and update channel, (3) log export, (4) administrator accounts, (5) backup store. Fictional system; no data.

Each control needs a test. Table 24.1 assigns a control, an observable test and an owner to each asset. **The key assumption of this chapter: a control counts only if a test shows that it works, with the expected result written down before the test is run.** A policy that says “access is restricted” is not a control until someone has tried a forbidden access and seen it refused.

Table 24.1: Controls, tests and owners for the fictional second-look service. Fictional roles; the tests of the first four rows are demonstrated in the lab with synthetic data.

Asset	Goal	Control	Observable test (expected result)	Owner
Images and reports	Confidentiality	Least privilege: each role reads only what its task needs; provider support has no read access to report text; deny by default	Support role requests a report: refused, refusal logged without content	System owner, IT security
Model file and configuration	Integrity	Release record with SHA-256 checksum from the provider's release process; server refuses to load a file whose checksum differs	Load a modified file: refused	Provider (release), clinical IT (installation)
Alert and reporting	Availability	Regular reporting pathway as fallback; backups with checksums; restore procedure	Restore from a verified copy; synthetic test cases give the expected outputs	System owner, radiology lead
Logs	Integrity, minimal content	Event logs without health data; entries chained by hashes	Edit one entry: chain check fails at that entry; scan finds no forbidden fields	IT security, data protection officer
Keys	Confidentiality	Encryption at rest and in transit; keys in a separate key store with named custodians and rotation	Restore succeeds only with the key from the key store; key rotation rehearsed	IT security

Where the duties come from. The GDPR requires security appropriate to the risk, including the ability to ensure the ongoing confidentiality, integrity, availability and resilience of systems, the ability to restore availability and access to personal data in a timely manner after an incident, and a process for regularly testing the effectiveness of the measures (Art. 32(1)(b)–(d)) [European Parliament and Council of the European Union (2016); Law]. For medical device software, the MDR requires development according to the state of the art including information security, and minimum IT security requirements set by the manufacturer, including protection against unauthorised access (Annex I, Sections 17.2 and 17.4) [European Parliament and Council of the European Union (2017); Law]. High-risk AI systems must allow automatic logging, and deployers must keep the logs under their control for at least six months unless other law provides otherwise (AI Act Art. 12 and 26(6)); for devices these duties apply from 2 August 2028 [European Parliament and Council of the European Union (2024); European Parliament and Council of the European Union (2026); Law]. The German Federal Office for Information Security (Bundesamt für Sicherheit in der Informationstechnik, BSI) recommends protecting the integrity of trained models with cryptographic measures and suitable monitoring and logging (measure M5) [Federal Office for Information Security (BSI) (2025); Guidance]. NIST describes the AI supply chain as an attack surface and names the basic check: the provider publishes cryptographic hashes, and the recipient verifies them; models should be treated as untrusted system components [Vassilev et al. (2025),

Sec. 3.2; Guidance]. The same idea appears in NIST's secure development framework as a mechanism for verifying software release integrity, for example by posting hashes or signing code (practice PS.2) [Souppaya et al. (2022); Guidance].

Checksum and signature. A checksum such as SHA-256 is a fingerprint of a file: if a single byte changes, the fingerprint changes. It shows that the installed file is the approved one only if the expected value comes from a trustworthy source, such as a signed release record. A digital signature binds that value to the key of whoever released it. The lab demonstrates the checksum part; signature verification uses separate tools and key management and is described here only conceptually.

Log events, not content. The incident began because a debugging setting wrote complete requests, including report text, into a support log that left the hospital. A useful log records that something happened, not what the patient has: time, a random request number, the model version's checksum, whether an alert was shown, the duration and an error code. The link between request number and examination stays inside the image archive. Logs can be made tamper-evident by chaining: each entry contains the hash of the previous one, so that a deleted or edited entry breaks the chain.

Recovery is a test, not a promise. NIST's contingency planning guide puts it briefly: testing validates recovery capabilities [Swanson et al. (2010), Executive Summary; Guidance]. A restore that has never been tried is an assumption. A recovery test uses synthetic cases with known expected outputs, so that it can be run at any time without touching patient data.

Question for the workshop: The nightly backup ran at 02:00 on Monday, after the wrong model file was installed on Sunday. What does your restore procedure do with that backup, and who decides which older copy is used?

24.3 Lab

The lab demonstrates the three control tests with synthetic objects only. Nothing is written to disk and nothing connects to a network; in practice the same functions work on files (`digest(file = ...)`).

Test 1 – an unauthorised read is refused and logged without content.

```
# Role-based permissions: TRUE = may read (deny by default for everything not listed)
permissions <- data.frame(
  role           = c("radiologist", "ed_clinician", "inference_service", "vendor_support",
        "it_admin"),
  image_pixels   = c(TRUE, TRUE, TRUE, FALSE, FALSE),
  report_text    = c(TRUE, TRUE, FALSE, FALSE, FALSE),
  model_file     = c(FALSE, FALSE, TRUE, FALSE, TRUE),
  service_events = c(FALSE, FALSE, FALSE, TRUE, TRUE)
)
allowed_log_fields <- c("time", "request_id", "role", "resource", "decision")

request_access <- function(role, resource, time, log) {
  granted <- role %in% permissions$role && resource %in% names(permissions)[-1] &&
    isTRUE(permissions[permissions$role == role, resource])
```

```
  entry <- data.frame(time = time, request_id = sprintf("R%04d", nrow(log) + 1), role =
        role,
                      resource = resource, decision = if (granted) "granted" else
        "refused")
  list(granted = granted, log = rbind(log, entry))
}

log <- data.frame(time = character(), request_id = character(), role = character(),
                  resource = character(), decision = character())
r1 <- request_access("vendor_support", "report_text", "2026-11-09 07:52", log)
r2 <- request_access("radiologist", "report_text", "2026-11-09 07:53", r1$log)
r3 <- request_access("unknown_account", "image_pixels", "2026-11-09 07:54", r2$log)
r3$log
```

```
#>               time request_id            role     resource decision
#> 1 2026-11-09 07:52      R0001  vendor_support  report_text  refused
#> 2 2026-11-09 07:53      R0002     radiologist  report_text  granted
#> 3 2026-11-09 07:54      R0003 unknown_account image_pixels  refused
```

```
stopifnot(!r1$granted,                                  # expected: refused
          r2$granted,                                   # expected: granted
          !r3$granted,                                  # unknown accounts are refused
          identical(names(r3$log), allowed_log_fields), # the log holds no content fields
          sum(r3$log$decision == "refused") == 2)
```

Test 2 – a wrong model version is detected before it is used. The “model file” is a short text with the coefficients of a synthetic logistic model. The release record holds its SHA-256 checksum.

```
library(digest)
# A model file as plain text: identical bytes on every computer, so the checksum is
        reproducible
model_text <- function(coef) paste(sprintf("%s=%.4f", names(coef), coef), collapse = "\n")
sha256 <- function(x) digest(x, algo = "sha256", serialize = FALSE)

approved_coef  <- c(intercept = -2.2, cortical_line = 3.1, soft_tissue_sign = 0.8)
approved_file  <- model_text(approved_coef)
release_record <- data.frame(version = "1.2.0", sha256 = sha256(approved_file))

# On Sunday a different file was installed: one coefficient differs
installed_file <- model_text(replace(approved_coef, "cortical_line", 2.6))

load_model <- function(file, record) {
  if (sha256(file) != record$sha256) stop("refused: checksum does not match release ",
        record$version)
  vals <- as.numeric(sub(".*=", "", strsplit(file, "\n")[[1]]))
  setNames(vals, sub("=.*", "", strsplit(file, "\n")[[1]]))
}

data.frame(file = c("approved", "installed"),
           sha256_first_16 = substr(c(sha256(approved_file), sha256(installed_file)), 1,
        16),
           matches_release = c(sha256(approved_file), sha256(installed_file)) ==
        release_record$sha256)
```

```
#>        file  sha256_first_16 matches_release
#> 1  approved f70d17b16faca3e3            TRUE
#> 2 installed 3ac397ad14f726b2           FALSE
```

```
outcome <- tryCatch(load_model(installed_file, release_record), error = function(e)
        conditionMessage(e))
outcome
```

```
#> [1] "refused: checksum does not match release 1.2.0"
```

```
stopifnot(sha256(approved_file) == release_record$sha256,
          sha256(installed_file) != release_record$sha256,
          identical(outcome, "refused: checksum does not match release 1.2.0"))
```

The same test runs at three moments: when a release is received, before the server loads a model, and on a schedule while it runs. In the case, the scheduled check found the deviation.

Limited, tamper-evident logging. Each event entry stores the hash of the previous entry.

```
add_event <- function(chain, time, event, model_sha) {
  prev <- if (nrow(chain) == 0) "start" else chain$hash[nrow(chain)]
  hash <- sha256(paste(time, event, model_sha, prev, sep = "|"))
  rbind(chain, data.frame(time = time, event = event, model_sha = model_sha, prev = prev,
        hash = hash))
}
# Returns the first entry whose stored hash or link does not match; 0 = chain intact
verify_chain <- function(chain) {
  for (i in seq_len(nrow(chain))) {
    prev <- if (i == 1) "start" else chain$hash[i - 1]
    expected <- sha256(paste(chain$time[i], chain$event[i], chain$model_sha[i], prev, sep =
        "|"))
    if (chain$prev[i] != prev || chain$hash[i] != expected) return(i)
  }
  0L
}
chain <- data.frame(time = character(), event = character(), model_sha = character(), prev
        = character(),
                    hash = character())
chain <- add_event(chain, "2026-11-08 21:14", "alert_shown", sha256(installed_file))
chain <- add_event(chain, "2026-11-08 21:40", "no_alert", sha256(installed_file))
chain <- add_event(chain, "2026-11-09 07:50", "release_check_failed",
        sha256(installed_file))
transform(chain[, c("time", "event")], model_sha = substr(chain$model_sha, 1, 12), hash =
        substr(chain$hash, 1, 12))
```

```
#>               time                event    model_sha         hash
#> 1 2026-11-08 21:14          alert_shown 3ac397ad14f7 c0d150a49085
#> 2 2026-11-08 21:40             no_alert 3ac397ad14f7 0bf1005f7f06
#> 3 2026-11-09 07:50 release_check_failed 3ac397ad14f7 4254a19cddab
```

```
tampered <- chain
tampered$event[2] <- "alert_shown"            # someone edits the second entry afterwards
c(original = verify_chain(chain), tampered = verify_chain(tampered))
```

```
#> original tampered
#>        0        2
```

```
stopifnot(verify_chain(chain) == 0, verify_chain(tampered) == 2,
          !any(c("report_text", "patient_name", "accession_number") %in% names(chain)))
```

Test 3 – recovery from a verified copy, checked with synthetic cases. Three nightly backups exist. The newest was made after the wrong file was installed. The restore takes the newest copy whose checksum matches the release record and then runs two synthetic test cases whose expected probabilities were written down at release.

```
backups <- data.frame(
  taken  = c("2026-11-09 02:00", "2026-11-08 02:00", "2026-11-07 02:00"),
  file   = c(installed_file, approved_file, approved_file)
)
restore <- function(backups, record) {
  ok <- vapply(backups$file, sha256, character(1)) == record$sha256
  if (!any(ok)) return(NULL)                    # no verified copy: stay on the fallback
        pathway
  list(taken = backups$taken[which(ok)[1]], file = backups$file[which(ok)[1]])
}
restored <- restore(backups, release_record)
restored$taken
```

```
#> [1] "2026-11-08 02:00"
```

```
# Synthetic test cases with expected outputs fixed at release (probability of fracture)
test_cases <- data.frame(case = c("T1", "T2", "T3"), cortical_line = c(1, 1, 0),
        soft_tissue_sign = c(1, 0, 0),
                         expected = c(0.8455, 0.7109, 0.0998))
predict_case <- function(coef, cases) {
  plogis(coef[["intercept"]] + coef[["cortical_line"]] * cases$cortical_line +
           coef[["soft_tissue_sign"]] * cases$soft_tissue_sign)
}
model <- load_model(restored$file, release_record)
test_cases$observed <- round(predict_case(model, test_cases), 4)
test_cases$pass <- abs(test_cases$observed - test_cases$expected) <= 0.0005
# For comparison: what the wrong file would have produced
wrong <- setNames(as.numeric(sub(".*=", "", strsplit(installed_file, "\n")[[1]])),
        names(approved_coef))
test_cases$wrong_file <- round(predict_case(wrong, test_cases), 4)
test_cases
```

```
#>   case cortical_line soft_tissue_sign expected observed pass wrong_file
#> 1   T1             1                1   0.8455   0.8455 TRUE     0.7685
#> 2   T2             1                0   0.7109   0.7109 TRUE     0.5987
#> 3   T3             0                0   0.0998   0.0998 TRUE     0.0998
```

```
stopifnot(restored$taken == "2026-11-08 02:00", all(test_cases$pass),
          any(abs(test_cases$wrong_file - test_cases$expected) > 0.0005))
```

The restore skipped the newest backup and used the copy from Sunday 02:00. The synthetic cases pass within the tolerance. The last column shows why both checks are needed: with the wrong file, all three cases would have produced the same alert decisions at a threshold of 0.5, and only the probabilities reveal the difference. A test on the alert decision alone would have passed a wrong model.

Recovery test protocol (expected result). The lab is the core of a protocol that can be rehearsed every quarter without patient data:

Table 24.2: Recovery test protocol for the second-look service. Fictional roles; uses only synthetic test cases.

Step	Action	Expected result	Responsible
1	Switch the alert off; confirm that all radiographs reach the regular reporting worklist	No examination without a reporting task	Radiology lead
2	Block remote support access; secure logs and the installed file unchanged	Access refused; copies with checksums stored	IT security
3	Verify backups against the release record, newest first	First matching copy identified; non-matching copies kept as evidence	Clinical IT
4	Restore and run the synthetic test cases	All cases within tolerance	Clinical IT
5	Re-enable the alert	Signed by the system owner after steps 1–4	System owner

24.4 Counter-check

Change one assumption and predict the outcome first. *Version 1:* the hospital keeps only one backup, the newest. *Version 2:* to help with debugging, the provider asks for the full request, including the report text, to be written to the event log.

```
# Version 1: only the newest backup, taken after the wrong file was installed
only_newest <- restore(backups[1, ], release_record)
is.null(only_newest)
```

```
#> [1] TRUE
```

```
# Version 2: a log schema with a content field
debug_log <- cbind(r3$log, report_text = "<synthetic report text>")
forbidden <- setdiff(names(debug_log), allowed_log_fields)
forbidden
```

```
#> [1] "report_text"
```

```
stopifnot(is.null(only_newest), identical(forbidden, "report_text"))
```

In version 1 the restore finds no verified copy and returns nothing. That is the correct behaviour: the service stays off and reporting continues on the regular pathway until a verified release is obtained again from the provider. A procedure that restored the newest backup without a checksum would have reinstalled the wrong model and reported success. In version 2 the schema check fails before any entry is written. Debugging with real report text is a request for patient data and belongs in the data-protection review (Chapter 23), not in a log setting.

24.5 Consequences for patients and operations

A failed control test changes the plan. The version check failed and, in the counter-check, a single-backup strategy would have failed as well. Table 24.3 shows the resulting changes.

Table 24.3: Deployment plan for the second-look service before and after the incident. Fictional plan.

Element	Plan before the incident	Plan after the incident
Re-enabling the alert	When the provider says it is fixed	After a verified restore, passing synthetic test cases and the system owner's signature
Remote support	Permanent access	Access for a named ticket and a fixed time window; no read access to report text
Logging	Debug level in production	Event schema without content fields; schema checked at every release; hash-chained entries
Backups	Newest copy overwrites older ones	Several dated copies, each with a checksum; at least one copy that cannot be overwritten from the server
Version check	At installation	At installation, before loading and every hour
Rehearsal	None	Quarterly recovery test with the protocol of Table 24.2

For patients, switching the alert off costs a second look but no report, because the regular pathway never depended on the tool. That is the design requirement behind every fallback: the tool may add safety, but its absence must not remove any. **Ethics** (Chapter 21): patients whose report text was exposed need an explanation and a contact person. **Regulation** (Chapter 22): a wrong version of a medical device in use is a question for the person responsible for device safety and possibly for vigilance reporting. **Data protection** (Chapter 23): the support log is a possible personal data breach with its own clock.

ⓘ The incident from the security perspective

How is the data leak contained, how is the integrity of the model checked, and how does reporting continue during the outage? Contain: block the access path, stop the debug logging, and ask the provider to secure and then delete the affected log. Preserve: keep copies of the logs and of the wrong model file with their checksums, so that the analysis is possible later. Continue: regular reporting without the alert. Recover: restore the approved version from a verified copy and test it with known synthetic cases before re-enabling. The technical restart ends with the system owner's signature; the data-protection assessment ends with a documented decision on notification and communication (Chapter 23). The two have different triggers, owners and end points, and a good handover names both and the person who decides on further clinical use.

24.6 Look up and transfer

Reviewer question. Which exact model version produced the results in the paper, and how could a reader verify it? A version number without a checksum or release record, or code without the trained model's fingerprint, does not allow an independent check (Methods in R: Writing a report – reproducibility).

Self-test. A research group keeps its trained model as `model_final.rds` on a shared drive. A colleague retrains the model with corrected labels and saves it under the same name. Three months later the group cannot reproduce the numbers in its submitted manuscript. Which two controls from this chapter would have detected the problem, and which test would you run from now on?

Answer

Annotated sources.

- European Parliament and Council of the European Union (2016) – Art. 32 GDPR: confidentiality, integrity, availability, timely restore and regular testing.
- European Parliament and Council of the European Union (2017) and European Parliament and Council of the European Union (2024) – IT security requirements for device software and logging duties for high-risk AI systems.
- Federal Office for Information Security (BSI) (2025) – BSI overview of generative AI risks; measure M5 on the integrity of models and training data.
- Vassilev et al. (2025) – NIST taxonomy of attacks on AI systems; Section 3.2 on supply-chain attacks and hash verification.
- Souppaya et al. (2022) – NIST Secure Software Development Framework; practice PS.2 on verifying release integrity.
- Swanson et al. (2010) – NIST contingency planning guide; why recovery capabilities must be tested.

24.7 Further reading

- Chapter 4 – a reproducible R project in which file versions and outputs can be traced.
- Chapter 20 – stop criteria, fallback and restart in the monitoring plan.
- Chapter 23 – the data-protection side of the same incident.
- Chapter 25 – attacks specific to learning and generative systems, from prompt injection to poisoned artefacts.
- Scientometrics in R: Reproducible pipelines – pipelines that record inputs and outputs, the basis of any release record.
- Scientometrics in R: Reproducibility, RRIDs, and badges – persistent identifiers that make artefacts citable and checkable.
- CTTIR Courses: R, RStudio, Quarto, and renv toolchain – pinning software versions so that a restored model runs in the same environment.

- Methods in R: Common errors – Git – version control mistakes that make the “approved version” impossible to find.