



GHOST
PROTOCOL

HA

THE FIELD LIBRARY

DEFENCE

BUILD IT TO HOLD

Hardening and Architecture

Secure by design, hardening the host and network, identity and access, application and cloud, and defence in depth.



Ghost Protocol (Pvt) Ltd · Colombo · ghosts.lk

Security internship training · The Field Library · Edition 1 · 2026

THE GHOST PROTOCOL FIELD LIBRARY · DEFENCE

Hardening and Architecture

Secure by design, hardening the host and network, identity and access, application and cloud, and defence in depth.

Ghost Protocol (Pvt) Ltd · Colombo

The Field Library, Edition 1 · 2026

Copyright 2026 Ghost Protocol (Private) Limited. All rights reserved.

Sold as a PDF for the buyer's own reading. Please do not share or resell the file.

The library

The Field Library is a whole shelf, not one book. The Core Path is the spine everyone reads. The rest is depth and breadth, taken on as the work calls for it. The shelf is complete now, and you are reading the one marked here.

A · THE CORE PATH

O The Oath BUILT

I Groundwork BUILT

II The Web BUILT

III The Wider Field BUILT

IV The Work BUILT

V The Craft BUILT

B · DEEP TRACKS

AI and Agents BUILT

Reading Code BUILT

Internal and Active Directory BUILT

The Web, Deeper BUILT

Cloud and Containers BUILT

Mobile and APIs BUILT

Reverse Engineering BUILT

Hardware and the Edges BUILT

C · THE OPERATIONS

Red and Purple Team BUILT

OSINT and Recon BUILT

Report Writing Mastery BUILT

The Consulting Craft BUILT

Bug Bounty BUILT

D · THE OTHER SIDE

Detection and Response BUILT

Threat Hunting and Forensics BUILT

Hardening and Architecture YOU ARE HERE

E · THE GROUND RULES

The Law, Deeper BUILT

Standards and Compliance BUILT

Ethics and Disclosure BUILT

F · COMPANIONS

The Reference Pack BUILT

The Lab Workbook BUILT

Case Files BUILT

The Bestiary BUILT

The Glossary and the Ledger BUILT

Contents

- Opening 5
- 1** Secure by Design 6
- 2** Hardening the Host and Network 11
- 3** Identity and Access Architecture 16
- 4** Application and Cloud Hardening 20
- 5** Defence in Depth and Assume Breach 25
- Worked answers 31

The other direction



Blink had a list of new things to break and wanted to go hunting bugs again. Breaking was the fun part, and the eager cursor knew it.

BLINK "More targets. Let us find more holes."

SOL "Turn it around once more tonight. You spent most of this library learning to break things, and the last two books taught you to watch for an attacker and to hunt the one who slipped past. This one comes earlier than both: build the thing so there is less to catch. Breaking is exactly what makes you able to build things that hold. Every weakness you learned to exploit is a decision someone made badly at design time. The best defender in the room is the one who already knows, in their bones, how the attack would go. That is you now."

Worm's eye settled not on a bug but on a diagram, on the place a control should have been and was not.

WYRM "design it in."

The pages ahead are the builder's craft seen through the breaker's eye: threat modelling, hardening the host and the network, getting identity right, and the one idea under all of it, that no single wall is ever trusted to hold alone.

CHAPTER ONE

Secure by Design

The last two books taught you to catch an attacker and to hunt one. This one starts earlier: you read the design before the code exists, name what can go wrong, and put the control where it belongs instead of bolting it on at the end.

A defender who has never attacked guesses at what matters. You are not guessing any more. The offensive volumes taught you where software leaks, and Detection and Response and Threat Hunting and Forensics showed you what an intrusion looks like from inside, which means you can now read a design the way the attacker reads the running system, before a line of it is written. That is secure design in one sentence: you spend the cheap hour at the whiteboard so you do not spend the expensive week in the incident call. This shelf is defensive so that the attacker finds less, and the saving starts at the design, where a decision costs a sentence to change.

Bolting on is the default, and it is the expensive one

Left alone, security arrives last. The feature works in the demo, the launch date is set, and someone asks about authentication on the Thursday before. Now the control has to fit around decisions already made: the data model that mixed one tenant's rows with another's, the internal service that trusts anything on the network, the admin endpoint that was never meant to be reached from outside but is. Each of those is a sentence at the design stage and a migration under load later. The order you do the work in decides the price.

Designing security in means the opposite habit. Before the schema is fixed you ask who the users are and what each of them must never reach. Before the services talk you decide which of them trust each other and which do not. Before the first endpoint ships you know what it returns to a caller who is lying about who they are. None of this is extra work invented to slow you down. It is the same work, done when it is still a decision rather than a repair. And a control put off to "later", with no owner and no date against it, is not planned, it is hoped for, which is how the access check everyone agreed to add before launch is the one missing after it.

Threat modelling: Four questions, asked out loud

Threat modelling sounds like a ceremony and is really a conversation held over a drawing. Adam Shostack's framing reduces it to four questions, and they are the whole method: what are we working on, what can go wrong, what are we going to do about it, and did we do a good enough job. You can run it on a napkin for a single endpoint or over a week for a platform. The questions do not change with scale.

What are we working on. Draw the system as data moving between parts: the browser, the API, the database, the queue, the third-party it calls. Mark where the data crosses from one zone of trust into another. That crossing is a trust boundary, the same boundary Volume II taught you to hunt from the

outside. Here you are drawing it on purpose so you can see every place untrusted input reaches something that trusts it. Most real findings sit on a boundary someone forgot they had drawn.

What can go wrong. This is where having attacked pays off, and where a checklist keeps you honest so you do not only list the attacks you happen to find interesting. The checklist most teams use is STRIDE.

STRIDE, the six questions you ask of every boundary

STRIDE is Microsoft's threat-classification scheme, in use since the late 1990s. It is six categories, and each one is the negative of a property you wanted. You walk each element and each data flow in your drawing and ask all six. The value is not the acronym, it is that six prompts catch the class you would have walked past.

CATEGORY	THE THREAT	PROPERTY IT ATTACKS
Spoofing	Pretending to be someone or something you are not	Authentication
Tampering	Changing data or code that should not change	Integrity
Repudiation	Denying you did a thing, with no record to counter it	Non-repudiation
Information disclosure	Reading what you should not read	Confidentiality
Denial of service	Making the system unavailable to real users	Availability
Elevation of privilege	Doing what your role is not allowed to do	Authorisation

Run it against one flow to feel it. A user uploads a profile photo. Spoofing: can someone upload as another user, is the session actually bound to the account. Tampering: can the file path be steered to overwrite something, is the stored filename attacker-controlled. Repudiation: if the upload is later disputed, is there a log that says who and when. Information disclosure: does the response leak the storage path, can one user read another's upload by guessing an id. Denial of service: is there a size cap, can a hundred-megabyte file per request exhaust the disk. Elevation of privilege: does the upload handler run with rights it does not need, can a crafted file become code the server runs. Six prompts, six candidate findings, on a feature that looked like it only moved a picture.

What are we going to do about it. For each thing that can go wrong you have four honest options: remove it (drop the feature or the field that carries the risk), mitigate it (add the control), transfer it (push it to a component built to carry it, an identity provider, a payment processor), or accept it with eyes open and a signature. Accepting a risk is a real and sometimes correct answer. Accepting it silently, because no one wrote it down, is how the risk you chose to live with becomes the breach no one chose.

Did we do a good enough job. Go back over the model and check that each threat has a decision against it, that the decisions are the controls the code will actually carry, and that nothing was marked done because the meeting ran out of time. This last question is the one teams skip, and skipping it turns the model into a document rather than a defence.

**SOL SAYS**

The point of the model is not the diagram, which no one reads again. It is the argument it forces you to have before the decisions harden. I have watched a thirty-minute threat model on a whiteboard delete a feature that would have been a year of patching, because once the trust boundary was drawn everyone in the room could see that the field simply should not exist. Draw it, argue it, write down what you decided and who decided it. Then the code has something true to be measured against, and "did we do a good job" has an answer instead of a shrug.

The principles that make controls cheap to design in

A handful of design principles, most of them set out in a 1975 paper by Saltzer and Schroeder, are the reason some designs are easy to secure and others fight you the whole way. They are worth holding in your head because they turn vague caution into specific decisions.

Least privilege. Every part runs with the smallest set of rights that lets it do its job, and nothing more. The photo handler needs to write to one bucket, not to the whole account. Decided at design, this is one line in a policy. Retrofitted, it means untangling what a broad role was quietly relied on to do. Chapter Three builds identity around this, and the Cloud volume applies it to a cloud tenant.

Fail-safe defaults. When something is not explicitly allowed, it is denied. A new route is unreachable until it is granted, a new field is private until it is published, a new rule starts closed. The opposite, default-open, means every thing you forget is a thing that is exposed. A system that fails closed turns your omissions into outages, which you notice, instead of into leaks, which the attacker notices first.

Complete mediation. Every access is checked, every time, at a point that cannot be skipped. A check that lives in the client, or only on the first request of a session, is a check the attacker steps around. The authorisation decision belongs in one place the request has to pass through, the same lesson Volume II taught about broken access control, read now from the building side.

Economy of mechanism and open design. Keep the security-critical part small enough to understand completely, and do not rely on the design being secret. If the safety of your system depends on an attacker not knowing how it works, you have a countdown, not a control. The algorithm is public; the key is the secret.

The attacker's eye, pointed at your own drawing

Everything in this chapter is the offensive skill you already have, run before the system exists instead of after. When you draw the data flow, you are doing reconnaissance on a map only you can see. When you walk STRIDE, you are enumerating as you would against a live target, except nothing is deployed to find. When you choose a control, you are asking what it costs the attacker, not whether it looks like security. A login form that can be brute-forced all night costs nothing; a rate limit plus a second factor costs real effort. You can only make that judgment because you have been the one doing the effort. Keep pointing that eye at your own designs and the attacker who comes later finds the cheap wins already gone.

■ EXERCISE

Take a small app that is entirely yours: a side project, a lab build, anything with a login and a database. On one page, draw it as data moving between parts and mark every trust boundary. Pick the two flows that cross a boundary and walk all six STRIDE categories against each, writing one concrete candidate threat per category even when it feels forced, because the forced one is where the surprise lives. For every threat, write which of the four responses you choose and why. You now have a threat model. Keep it, and when you next change that app, check whether your change added a boundary the model does not show.

■ VERIFY IT

A model claims a control exists. Prove the control does what the model says before you believe the box is ticked. If the design says "the upload handler cannot write outside its bucket", test it on your own build: try to make it write elsewhere and confirm it is refused. A control named in a diagram and a control that actually holds under a push are two different things, and only one of them stops an attacker. The threat model earns trust the same way a finding does, by someone reproducing the claim with their hands.

■ THE LINE

Threat modelling is drawing and argument, so it feels safe, and mostly it is. The line appears the moment the model points at a system that is not yours and you reach to confirm a guess by probing it. Modelling a client's architecture from their documents, in a scoped engagement, is the work. Testing it is a separate authorisation, in writing, exactly as Volume 0 lays out. A diagram that suggests a weakness is a hypothesis, not a permission. You do not cross from the whiteboard to the live system on your own say-so, ever, not even to be helpful.

■ ON YOUR FEET

A developer shows you a new internal reporting service and says it needs no authentication because it is only reachable from inside the network, and the network is trusted. You have two minutes before the design is signed off. Which STRIDE categories do you raise first, what is the single assumption in that sentence you most want to test, and what do you propose they change in the design rather than add to it afterwards?



WHAT WYRM KEEPS

Security designed in is the same work as security bolted on, done while it is still a decision. Draw the system, walk STRIDE on every boundary, decide a response for each threat, and check that you did.

- Threat modelling is four questions: what are we working on, what can go wrong, what will we do about it, did we do a good enough job. The last one is the one teams skip.
- STRIDE is six prompts per boundary: spoofing, tampering, repudiation, information disclosure, denial of service, elevation of privilege. Each is the negative of a property you wanted.
- Least privilege, fail-safe defaults, and complete mediation are cheap at design and costly to retrofit. Fail closed so your omissions become outages, not leaks.
- Every threat gets a named response: remove, mitigate, transfer, or accept with a signature. A risk accepted silently is the breach no one chose.