HAWK EYE ANALYSIS

From PowerShell to Platform

How a Script Became a Security Suite

The story of how Hawk Eye Analysis Tool began as a single PowerShell script and evolved into a C++23 cybersecurity platform — and the unconventional engineering decisions that made it possible.

The Beginning

Norton Power Eraser, but Make It Mine

Every product has an origin story, and ours starts with a simple observation: Norton Power Eraser was a brilliant concept trapped inside a limited experience.

For those unfamiliar, Norton Power Eraser is Symantec's aggressive, on-demand threat hunting tool. Unlike the always-on Norton 360 shield, Power Eraser is the tool you deploy *after* something has gone wrong — when a machine is compromised and you need to find and rip out whatever is hiding. It uses deliberately aggressive heuristics that trade false positive tolerance for detection depth. It will flag things that Norton's main product wouldn't touch.

The concept was right. The execution left room for improvement. What if a tool like that could present its findings in a way that actually helped the user understand what was happening on their machine? What if the investigation experience was as thoughtful as the detection engine behind it?

That question became the Hawk Eye Analysis Tool — HEAT.

Timeline

The road from curiosity to platform spanned nearly two decades of accumulated expertise and a single year of intense product development.

2007	**The Interest Begins** — A fascination with cybersecurity takes root. Over the following years, deep expertise accumulates across Windows internals, malware analysis, persistence mechanisms, network security, and threat intelligence — the knowledge base that would eventually power every detection heuristic in the Helios engine.
July 2025	**First Script Drafted and Executed** — The Hawk Eye Analysis Tool (HEAT) is born as a PowerShell script under the codename Orion. The proprietary heuristic engine — compound signal accumulation, LOLBin parent analysis, process masquerading detection, PE static analysis — is the direct result of 18 years of cybersecurity expertise and rigorous hands-on testing against live malware samples. The browser-as-UI innovation follows immediately.

Aug — Sep 2025	**Rapid Evolution** — Updates ship continuously. The engine rebrands from Orion to Helios. Botnet DynaTune replaces handcrafted IOC rules with dynamic threat feed analysis. The exclusion system, Relationship Engine, and Application Legitimacy Heuristic are added in response to real-world false positives. The React report UI matures from vanilla JS to a full component architecture. The DCI cloud intelligence pipeline goes live with ThreatFox integration.
September 2025	**The Ceiling** — It becomes clear that PowerShell has been pushed as far as it can go. The P/Invoke overhead, the interpreted execution, the lack of real-time UI, the two-part scan-then-report workflow — every architectural ambition is fighting the runtime. The decision is made to reinvent the platform entirely.
September 2025	**The Stack is Chosen** — After careful evaluation, the technology stack is selected: C++23 with MSVC 18 Insiders Preview for the engine, React with TypeScript for the UI, WebView2 for the native bridge, Glaze for JSON serialisation, and Profile-Guided Optimisation for production builds. The OrbitUI design system is conceived as the visual identity layer.
December 2025	**DHC v1.0 Launches** — Just before the new year, the first version of the Defender Hardening Console ships. It launches modestly — an interface to configure and tweak Windows Defender settings — but the C++23/React/WebView2 foundation is in place. The architecture that will support the entire Hawk Eye product suite is live. Everything that follows builds on this moment.
2026	**The Platform Expands** — DHC grows from a settings interface into a full security console with the Helios scanning engine, real-time network monitoring, and the radial OrbitUI interface. Helios Web Marshall, Talon, The Inspector Hawk, and AiDefender join the product suite. Hawk Eye Analysis becomes a platform.

A PowerShell Security Suite

HEAT was written entirely in PowerShell. That sentence alone should raise eyebrows among anyone who's worked with the language, because what we built inside it had no business running there.

The architecture was modular from the start. A main orchestrator script coordinated four specialised engine modules: a Dynamic Cloud Intelligence system for real-time threat feed integration, a multi-engine static analysis module with PE header parsing, a botnet heuristics engine for network behaviour analysis, and a full remediation engine with stability controls. An external JSON configuration file served as the threat intelligence knowledge base.

The scanning pipeline ran in three tiers. A Gentle Scan checked live processes and modules. An Elevated Scan added file system analysis and persistence enumeration. An Aggressive Scan activated the full multi-engine static analysis, PUP detection, driver auditing, and system tampering checks.

Concurrent Network Monitoring

The live network monitor ran **five parallel pipelines** inside a RunspacePool, coordinated through thread-safe concurrent collections — `ConcurrentBag`, `ConcurrentQueue`, `ConcurrentDictionary`. A collector pipeline grabbed TCP connections and UDP listeners. A resolver pipeline performed reverse DNS lookups. A DCI checker pipeline matched connections against the live threat intelligence database. A domain checker pipeline evaluated resolved names. A Whois pipeline enriched everything with ASN and geolocation data.

All of this ran in the background while the main scan proceeded. In PowerShell. With no native threading support.

PE Header Parsing from Raw Bytes

The static analysis engine read the first 1024 bytes of candidate files and manually walked the PE structure — checking the MZ magic bytes, following the e_lfanew pointer to the PE signature, parsing the COFF header for compile timestamps, extracting the optional header magic to determine PE32 vs PE32+, reading the subsystem type, and checking the resource directory RVA. All through `[System.BitConverter]` calls on a raw byte array.

This wasn't academic exercise. The compile timestamp check caught backdated malware. The subsystem check flagged console applications running silently. The resource directory check identified files stripped of icons and version information — a hallmark of tooling-generated malware.

WinTrust Signature Verification via P/Invoke

We went deeper than `Get-AuthenticodeSignature` — defining the `WINTRUST_FILE_INFO` and `WINTRUST_DATA` structures in inline C#, marshaling them to unmanaged memory, and calling `WinVerifyTrust` directly through platform invoke. A signature cache avoided redundant verification. This gave us the same verification path that native security software uses, running inside an interpreted scripting language.

Shannon Entropy for DGA Detection

The botnet heuristics engine included a Shannon entropy calculator for identifying algorithmically generated domain names. Subdomains longer than ten characters with entropy above 3.5 bits per character were flagged as potential DGA activity. A small function with significant detection value.

The UI Problem Nobody Else Solved

PowerShell has no real UI framework. Every other PowerShell security tool solves this the same way: console menus. A wall of terminal text, numbered findings, and a prompt — *Enter 1 to quarantine, 2 to skip, 3 to investigate.* One item at a time, sequentially, no ability to see the full picture, no way to change your mind about finding #3 after seeing finding #17, no way to compare two findings side by

side. Miss a number and you're scrolling through a terminal buffer trying to find what you passed on.

Why Not WPF?

PowerShell runs on .NET, so technically you can load the WPF presentation framework and parse XAML directly. In practice, this means writing your entire UI as a string literal inside a PowerShell script — no syntax highlighting, no designer support, no component model, and no error checking until runtime. One wrong character in a 2000-line here-string and the whole thing crashes with an inscrutable parser error.

And it gets worse as it grows. HEAT's UI had accordions, modals, tabbed views, sortable tables, checkbox state management, a remediation confirmation flow, firewall rule staging, toast notifications, a starfield particle system, and a glitch-text animated counter. Try expressing any of that in XAML-as-a-PowerShell-string. Event handlers wired through `.FindName()` calls that fail silently on a typo. UI updates from background scan threads requiring `Dispatcher.Invoke()` calls everywhere. It would have been brittle, painful, and limited.

We rejected the "correct" solution and found a better one.

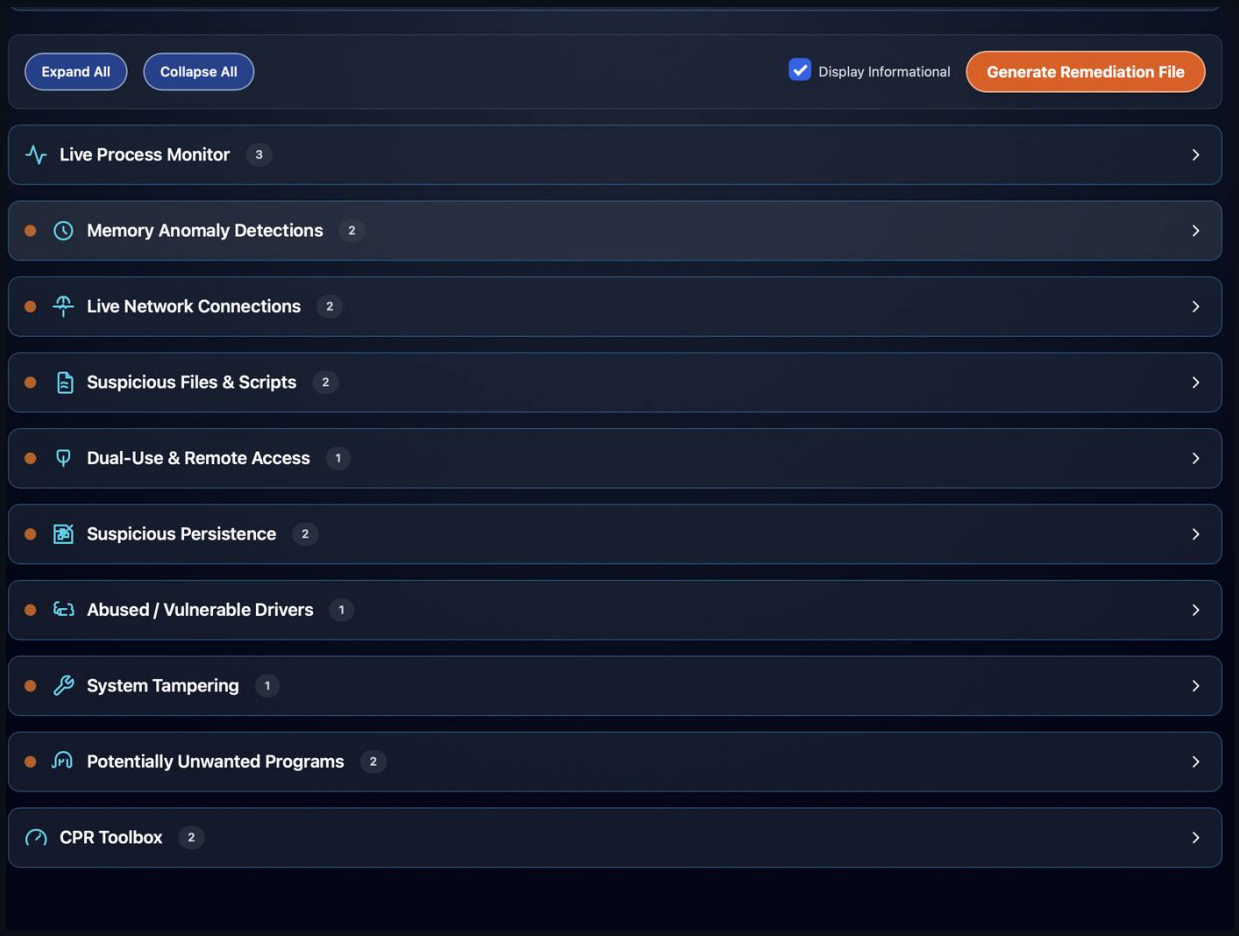
The Browser as a UI Framework

The scan generated a self-contained HTML file — a complete React single-page application with all scan data embedded as JSON literals. Tailwind CSS for styling. Font Awesome for icons. Google Fonts for typography. The file opened in the user's default browser and became a full interactive decision-making environment.

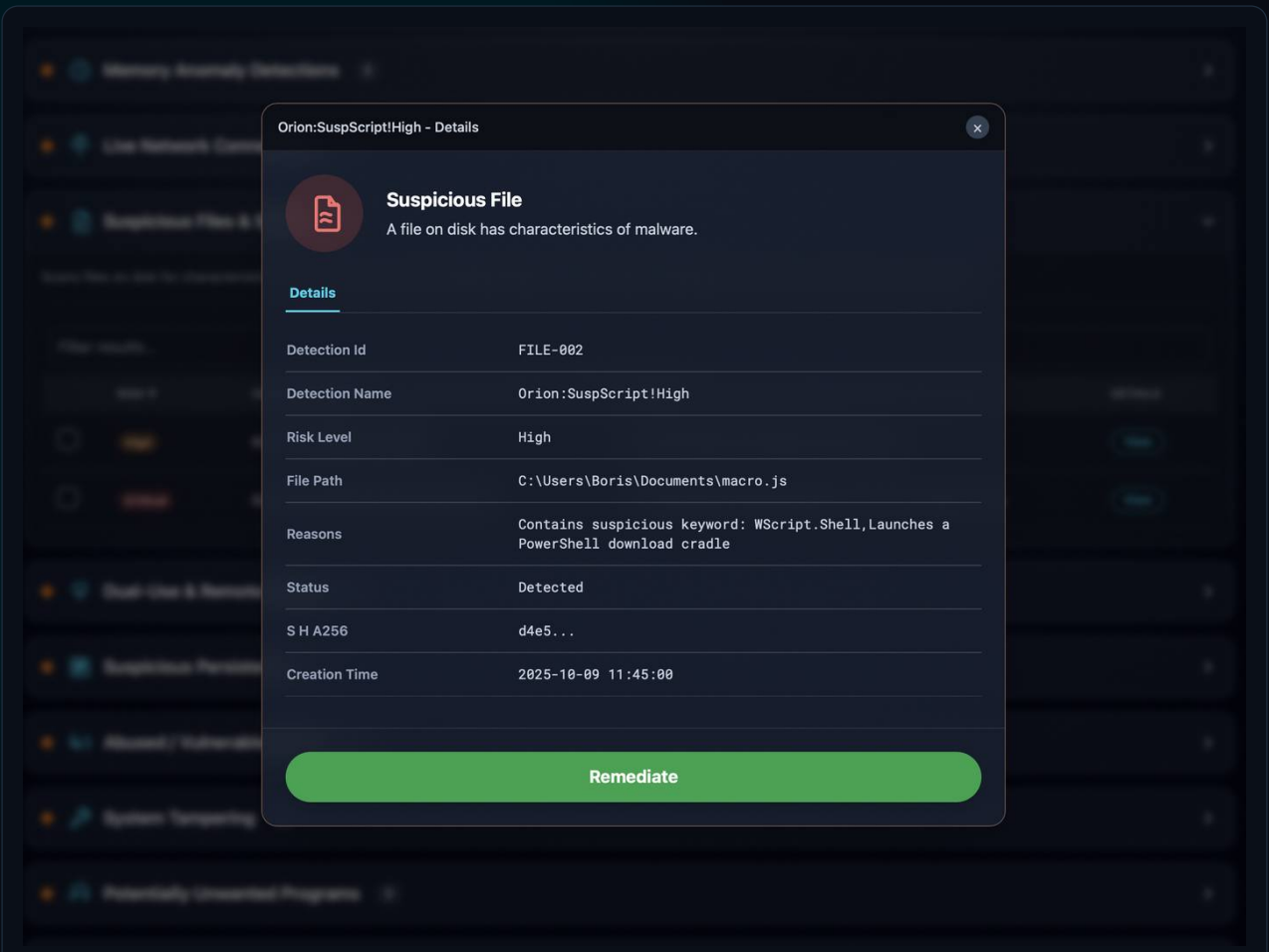
The user got glassmorphism containers with animated borders. Collapsible accordions for each finding category. Sortable, filterable tables. Detail modals with tabbed views for raw data, threat intelligence, and relationship analysis. A staged remediation workflow with confirmation dialogs. Toast notifications. An animated starfield particle system. A cyberpunk glitch-text counter.

HEAT gave the user something no console menu ever could: the ability to see everything at once, filter, sort, drill into details, read the threat intelligence, check the relationship analysis, stage firewall rules, manage exclusions, and then make all their decisions in a single batch. The user could look at a network finding, open the Whois tab, decide it was suspicious, check the box, then jump to a persistence finding referencing the same file path — building a mental picture of the infection chain across categories before committing to anything.

When the user finished reviewing, the browser exported a JSON file containing their decisions. The PowerShell script picked it up and executed the remediation. This was effectively a client-server architecture — the browser as frontend, PowerShell as backend, the filesystem as IPC, and a JSON file as the message bus.



The Helios-era report UI: collapsible accordions with pulsating detection dots, category counts, the 'Generate Remediation File' button, and the full findings hierarchy.



Helios detail modal for a suspicious script detection: glass-sphere icon, tabbed Details view, file hash, detection reasons, and the one-click Remediate button.

A poor man's WebView2 bridge. Nobody taught us this pattern. It doesn't exist in any documentation. We invented it because we refused to let the tooling dictate the experience.

The Detection Philosophy

The detection engine was built around the principle of compound heuristic analysis — combining multiple individually weak signals into high-confidence detections.

Layered Signal Accumulation

An unsigned binary is common. A file in the Downloads folder is normal. A console application is unremarkable. Missing version information happens. But an unsigned console application in the Downloads folder with missing version information, no PE resources, and a link to a Registry Run Key? That's malware.

The static analysis engine assigned weighted scores to each characteristic: 8 points for being unsigned, 4 for a risky location, 5 for console subsystem, 7 for missing version info, 6 for absent resources, 4 for

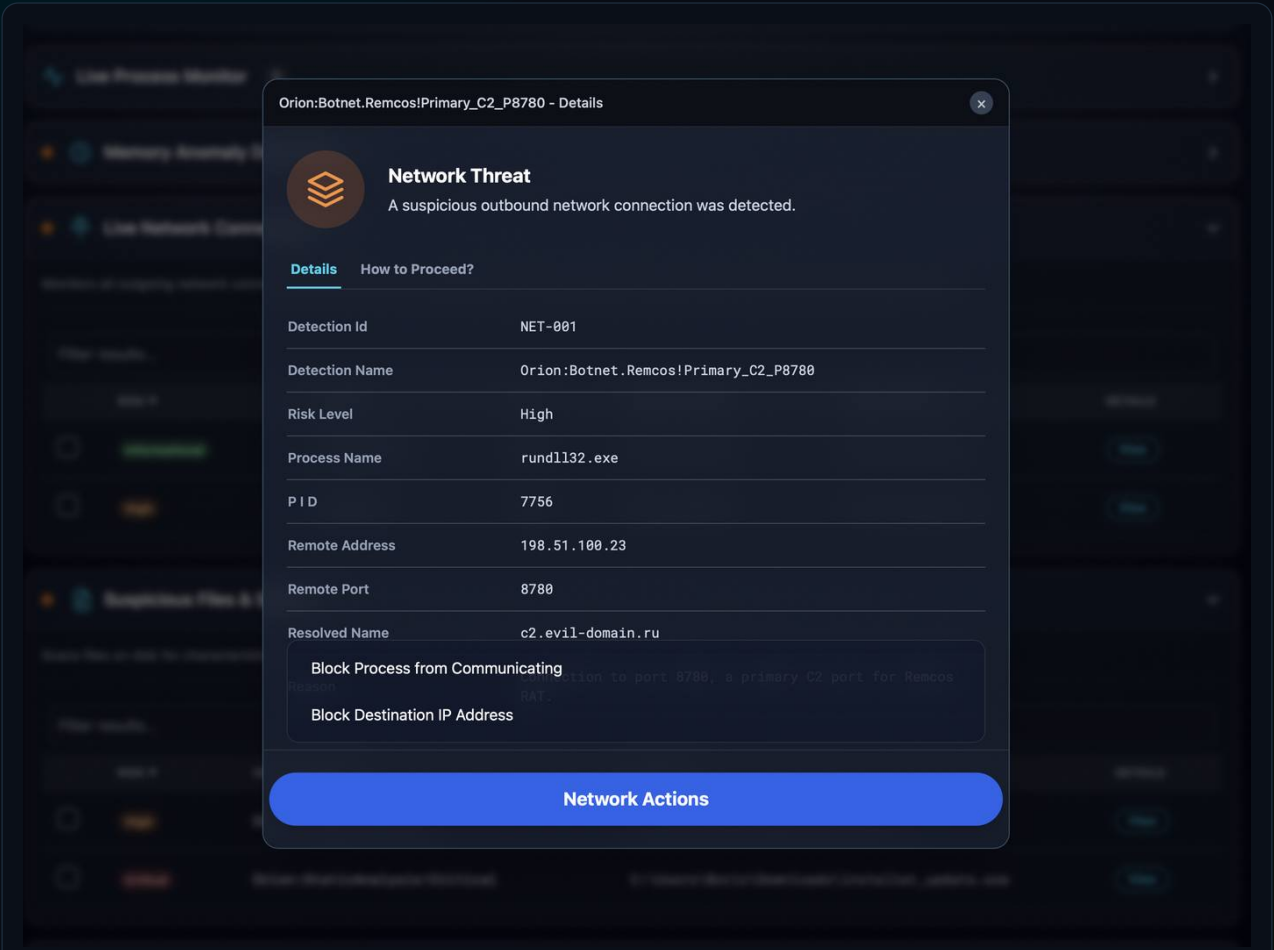
persistence linkage. Individual signals were noise. The combination was signal.

Dynamic Threat Profiling: Botnet DynaTune

The system we called Botnet DynaTune went beyond static IOC matching. Instead of maintaining a manually curated list of known-bad ports and hosting providers, the engine analysed the live ThreatFox threat intelligence feed to extract currently trending attack patterns.

It parsed recent IOCs to identify which ports were most frequently appearing in fresh C2 reports, which ASN ranges were being actively abused, and which malware families were using which infrastructure. This produced a real-time TTP map that the heuristic engine used for scoring.

A connection to port 8848 from an unsigned process was suspicious. But if port 8848 was also currently trending in the DynaTune feed as an active AsyncRAT C2 port, the confidence jumped significantly. And if the destination ASN was simultaneously trending as an abused hosting provider, the detection became near-certain — flagged as `Helios:NetworkHeur!AbusedCombo` at Critical severity.



Helios network threat modal showing a Remcos RAT C2 detection: process details, remote address, firewall action dropdown for blocking the process or destination IP.

False Positive Reasoning

Detection is only half the problem. We built a Relationship Engine that analysed a suspicious file's context to estimate the probability of legitimacy.

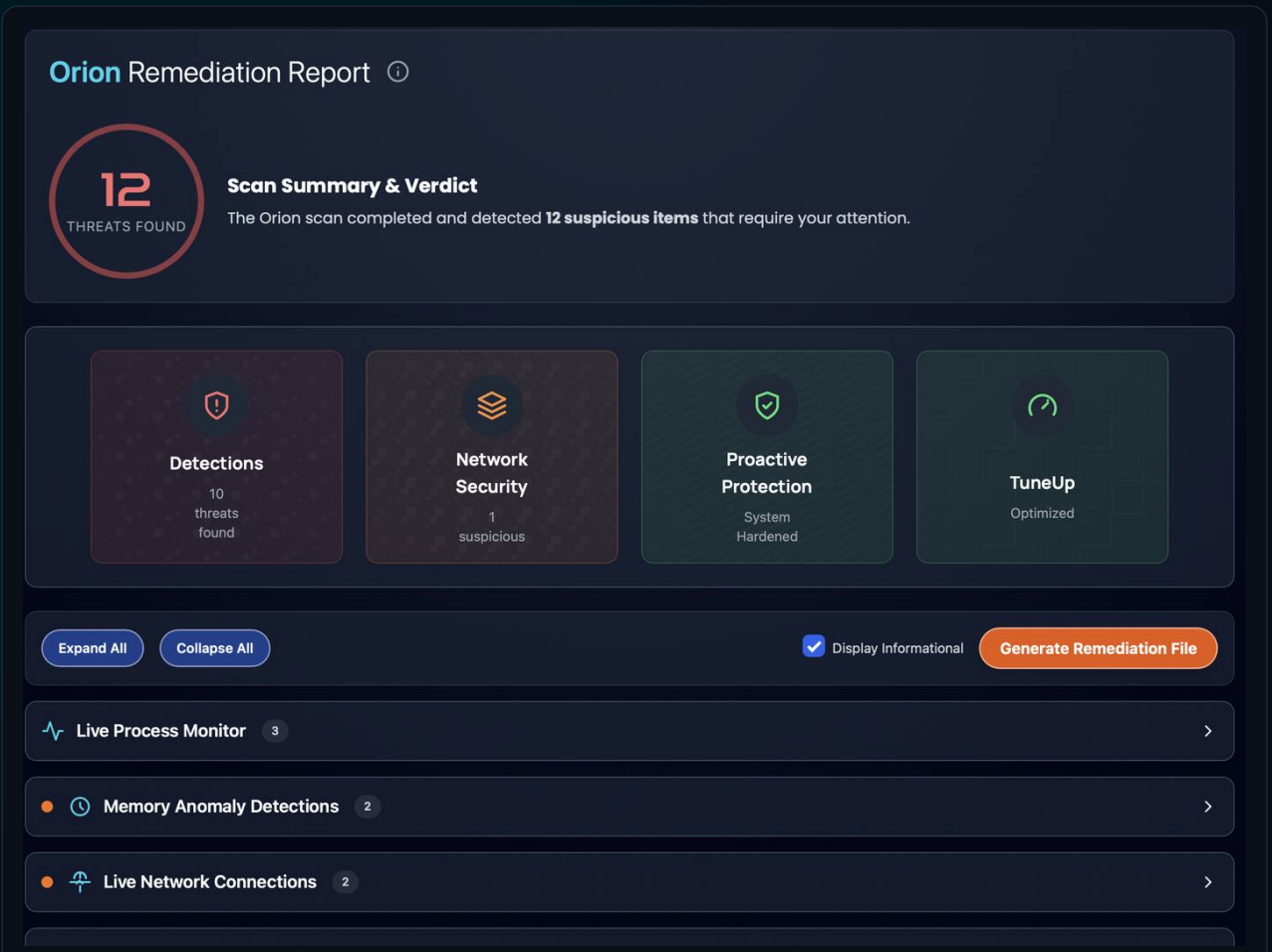
The engine checked whether the file's product name matched any registered installed program. It examined sibling files in the same directory — were they signed? Did they share a product name? Were their timestamps consistent with being installed together? It walked up the directory tree to find the installation root and matched folder names against the installed programs registry.

Design Influences: Norton, McAfee, and Finding Our Own Voice

Every product carries the DNA of its inspirations. HEAT drew from two distinct philosophies in the security industry — and eventually outgrew both.

The Norton 360 Dashboard

HEAT's report UI featured four status panels at the top of the page — Threats, Network, Tuneup, Harden — directly inspired by Norton 360's four-quadrant security dashboard. The panels used status-driven colour gradients (green for safe, amber for warnings, red for critical), animated background textures unique to each panel, and a hover interaction that revealed detailed breakdowns. In the code, they were literally named `n360-panel` and `N360Dashboard`.



The Orion-era report UI: N360-inspired dashboard panels with threat counter, status-driven colour gradients, and the accordion-based findings layout below.

The concept was sound: give the user an instant at-a-glance security posture before they engage with the detailed findings below. Most security tools dump you straight into a list of problems. The N360 panels acted as a triage layer — you see the overall health immediately and know where your attention should go. For a static HTML report, this was exactly the right design.

Between the Orion and Helios versions, the panels evolved from vanilla CSS with direct DOM manipulation to React 18 components with Tailwind CSS and a proper component architecture. The animated textures became inline SVG data URIs. The hover behaviour moved from CSS transitions to React group-hover state. They became more polished, more maintainable — and more our own.

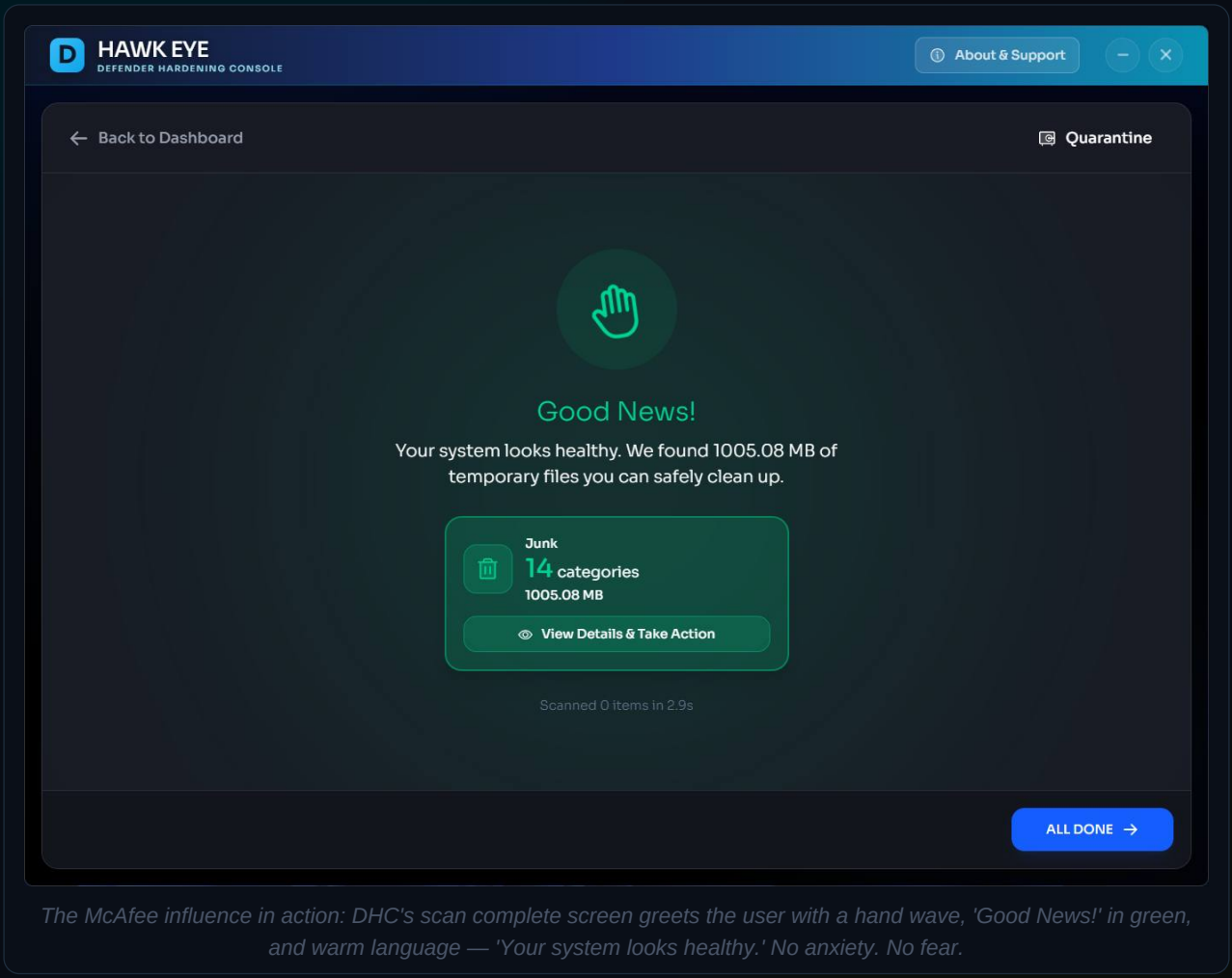
When the Defender Hardening Console arrived, the N360 panels were retired entirely. A live application doesn't need a summary dashboard at the top — the application itself is the status. They were replaced by a radial orbital interface where the system's health sits at the centre and every capability orbits around it. The product stopped referencing Norton's design language and found its own.

The McAfee Influence

Norton gave us the dashboard concept. McAfee gave us something more fundamental: the soul of the user experience.

McAfee's products never make a big deal out of things. No intrusive, heavy-handed alerts. No anxiety-driven language designed to make the user feel under siege. Where Norton and Kaspersky lean into urgency — red alerts, threat counters, aggressive warnings — McAfee treats security as something friendly and approachable. *You are protected.* Green checkmark. Warm language. The user should feel safe, not surveilled.

This philosophy runs through everything we build. The HEAT preloader doesn't say "WARNING: SCANNING FOR THREATS." It says "Bring the heat on malware" — playful, confident, not alarming. The scan summary says "System Clean" with a green ring, not "No Threats Detected" in clinical language. The CPR Toolbox isn't framed as "your system has problems" — it's "here are actions available to improve things." The farewell sequences in the DHC scanner. The hand wave greeting. The positive framing throughout.

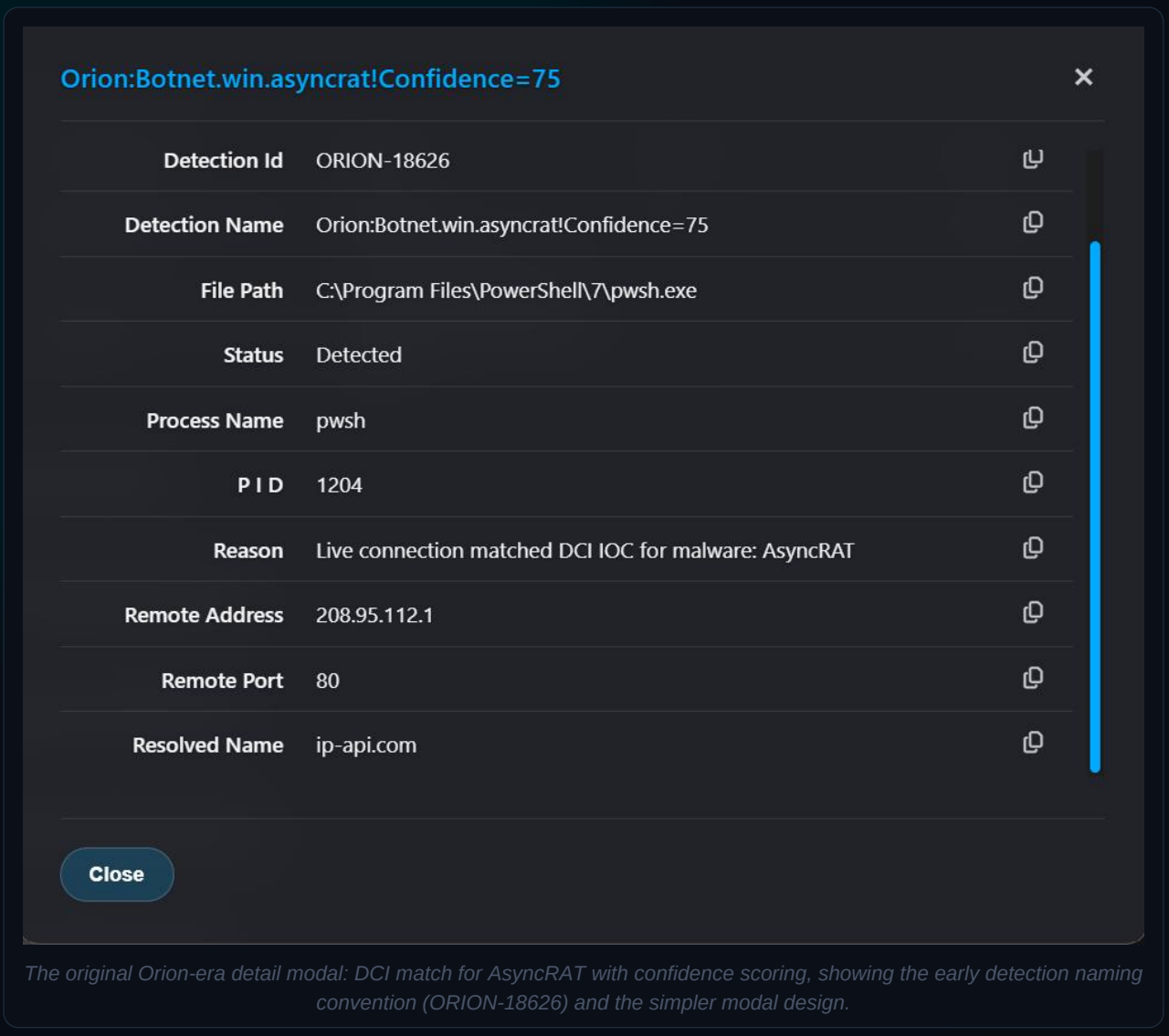


The McAfee influence in action: DHC's scan complete screen greets the user with a hand wave, 'Good News!' in green, and warm language — 'Your system looks healthy.' No anxiety. No fear.

This is a subtle but critical influence. Norton gave us the dashboard. Power Eraser gave us the forensic workflow. McAfee gave us the soul.

The Orion-to-Helios Evolution

The first version carried the codename Orion. Within the first few months, the engine was rebranded to Helios. But the rebrand marked a maturation of the detection philosophy.



The original Orion-era detail modal: DCI match for AsyncRAT with confidence scoring, showing the early detection naming convention (ORION-18626) and the simpler modal design.

The original Orion botnet heuristics were handcrafted signatures. Individual rules for individual RATs: AsyncRAT on port 8848, XWorm on port 9990, QuasarRAT on port 5552, Remcos on port 8780. Each was a manually written conditional. Effective, but brittle.

Helios replaced manual rules with dynamic discovery. DynaTune derived the trending ports and ASNs automatically from the live threat feed. The engine didn't need to know that AsyncRAT uses port 8848 — it discovered that port 8848 was currently active and cross-referenced it against process behaviour in real time.

Orion had no exclusion system. Every scan started from zero. Helios introduced full exclusion management backed by the Windows registry, with multi-strategy key derivation and UI support for staging exclusions during the review process.

Orion had no false positive reasoning. If a file scored above threshold, it was flagged. Helios added the Relationship Engine and Application Legitimacy Heuristic, allowing the engine to suppress findings for files that exhibited strong indicators of legitimate software.

Each of these changes came from real-world experience. Every improvement was a direct response to an observed limitation.

The Leap to C++23

After approximately four months of active development, HEAT had validated every important idea: the detection philosophy, the threat intelligence pipeline, the UX model, the remediation workflow. It had also reached the ceiling of what PowerShell could deliver.

The rewrite to C++23 wasn't a language preference — it was a liberation.

The Philosophical Shift

HEAT was a script you run after something goes wrong. DHC is a product that is always running. That's the single biggest change. HEAT launches, snapshots the system, generates a report, waits for the user to pick checkboxes, remediates, and exits. It's forensic — it looks at the crime scene after the fact. DHC with its persistent network monitor, real-time scoring caches, and WebView2 notification system is a continuous security posture. The user doesn't decide to scan. The system is always being observed.

HEAT started as Norton Power Eraser — an on-demand forensic tool. The N360 panels in its UI were a sign of where the product wanted to go. DHC completed that journey, evolving from Power Eraser's model to Norton 360's model — from an on-demand scanner to a persistent security console.

What C++23 Specifically Enabled

Direct Win32 API access eliminated P/Invoke overhead. `WinVerifyTrust` became a direct function call with stack-allocated structs. `GetExtendedTcpTable` replaced WMI queries. PE parsing became pointer arithmetic on memory-mapped files.

`std::expected` replaced try/catch blocks in hot paths, giving the compiler and Profile-Guided Optimisation the ability to reason about error handling as branching. `std::flat_map` provided cache-friendly contiguous memory layout for the lookup tables that the scoring engine hits thousands of times per evaluation cycle. `std::ranges` composed multi-step transformations into single-pass operations with zero intermediate allocation. `std::jthread` with cooperative cancellation via `stop_token` replaced the manual flag-polling pattern of the PowerShell runspace pool.

Additional C++23 features completed the picture: `constexpr` and `consteval` pre-computed detection name comparisons and risk level mappings at compile time — work that HEAT performed at runtime through string matching. `std::move_only_function` eliminated heap allocation overhead in the scoring callbacks that fire on every connection evaluation.

The Performance Impact

The gains varied dramatically by operation. PE header parsing and static analysis saw the largest improvements — 50 to 100x faster per file — because HEAT read bytes through .NET interop with bounds checking and GC pressure on every field extraction, while C++ uses direct pointer arithmetic on memory-mapped files. Across hundreds of candidates, this compounds from seconds into minutes saved.

Signature verification improved 10 to 20x on cache misses, eliminating the managed-to-native transition cost of every P/Invoke call. Network monitoring improved 20 to 30x per poll cycle by replacing WMI queries with direct `GetExtendedTcpTable` calls that return raw memory buffers of connection structs — no serialisation, no object creation, no garbage collection.

The scoring and enrichment pipeline is where PGO earned its keep. The hot paths through exclusion checks and heuristic evaluation were reshaped so the common "not a threat, skip" path became the fall-through branch. In HEAT, PowerShell's interpreter evaluated every condition through its AST walker — no branch prediction to optimise. The difference: 30 to 50x for evaluation logic alone.

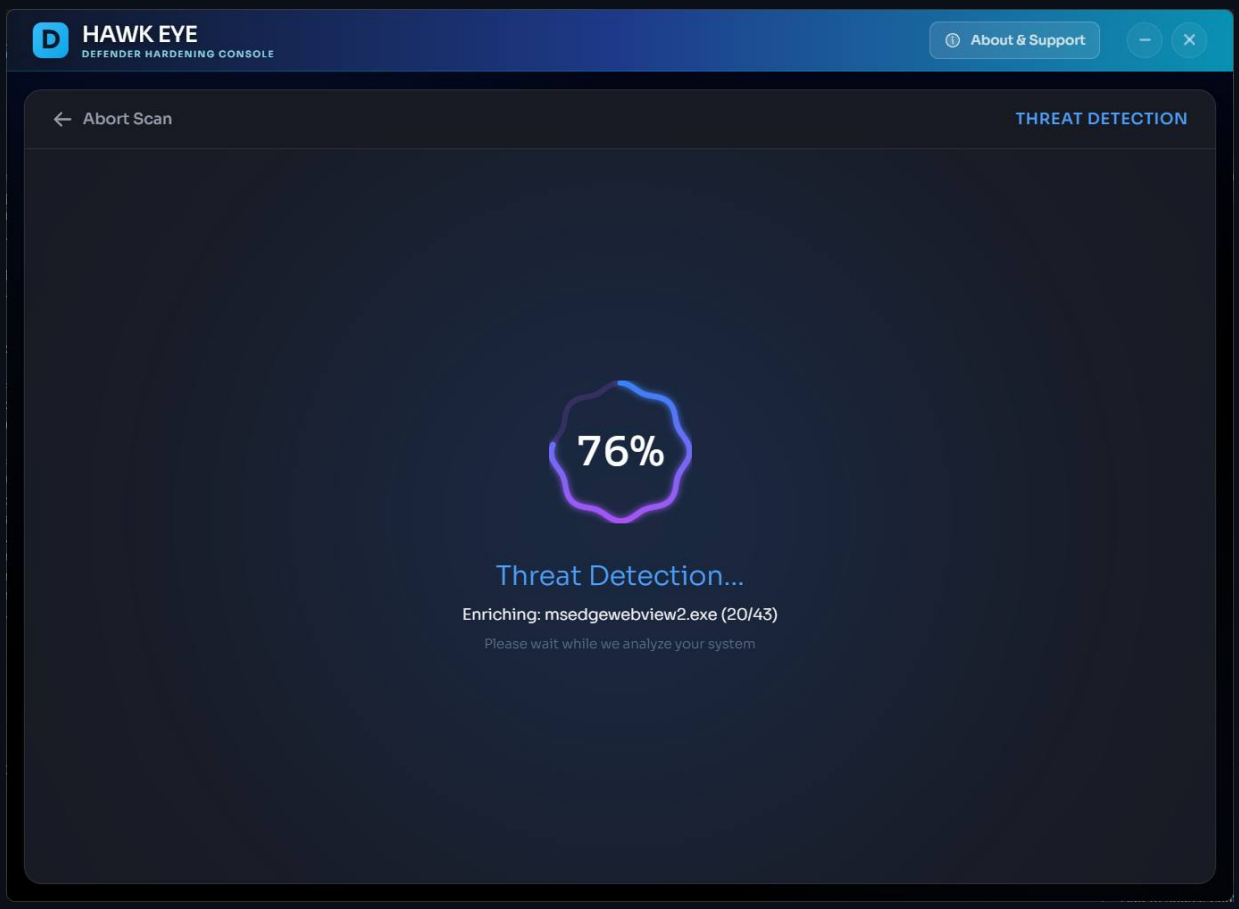
But the real win isn't any single operation. It's that the performance headroom changed what's architecturally possible. Continuous per-process scoring caches that update in real time, pre-computed enrichment results across thousands of connections, cloud KV reads optimised from thousands per hour to approximately fifty — none of these were conceivable in PowerShell.

From Report to Application

The most transformative change wasn't performance — it was the UI architecture. HEAT generated a static HTML report. The Defender Hardening Console is a native desktop application built with React running in WebView2.

The WebView2 bridge means a C++ detection event becomes a React state update in milliseconds. The scan starts from a button in the UI, progress updates in real time, findings appear as they're discovered, and remediation happens without the user ever leaving the application. There is no JSON file handoff. There is no alt-tab back to a terminal window. The seam between engine and interface doesn't exist.

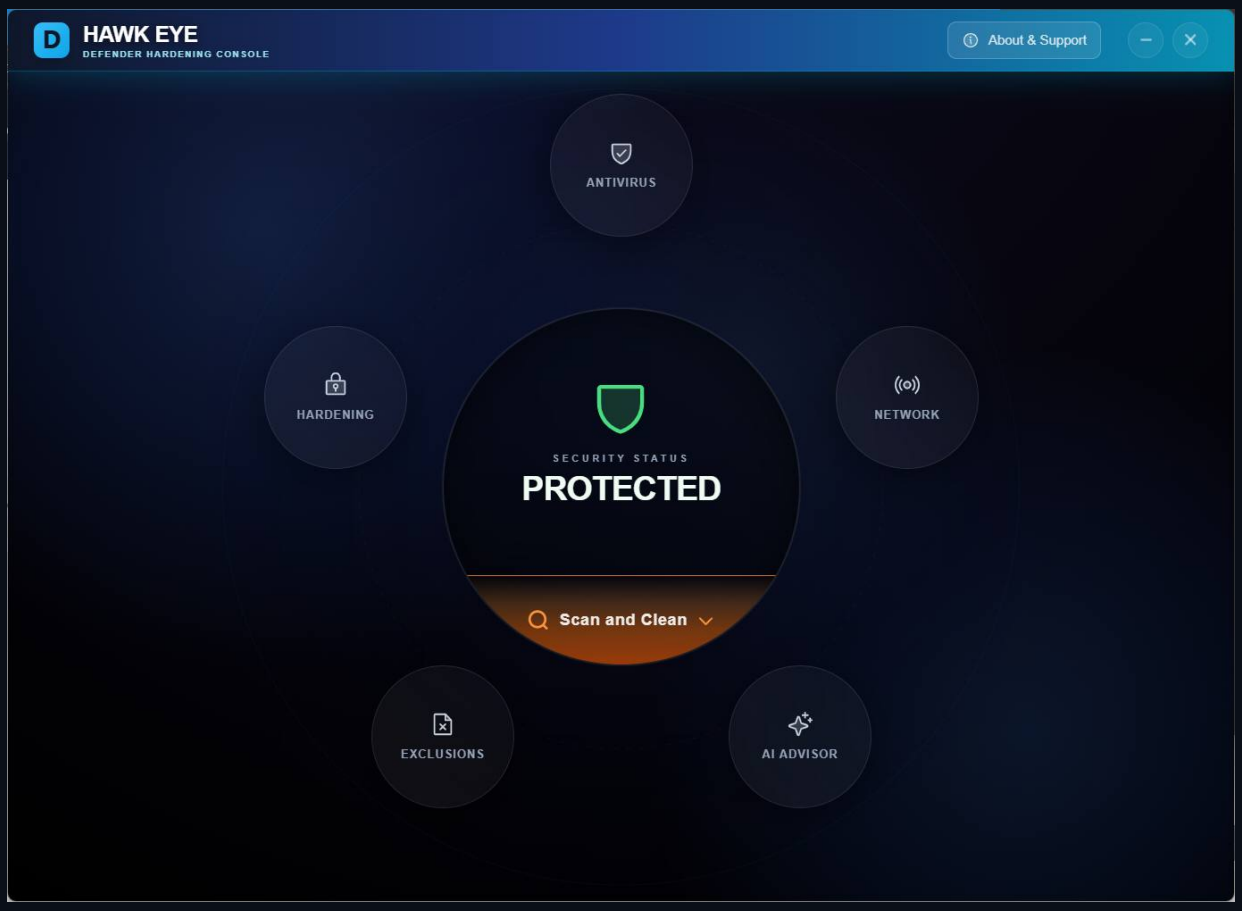
This changed what's possible in the UX. The scan progress ring with phase-specific subtitles, the directional slide transitions, the orbit circles hiding and reappearing, the cancel button that smoothly reverses everything — none of that works if the scan and the UI are separate processes. HEAT's JSON handoff was ingenious given the constraints. But it meant the scan was a black box to the user until it finished. DHC eliminated that gap entirely.



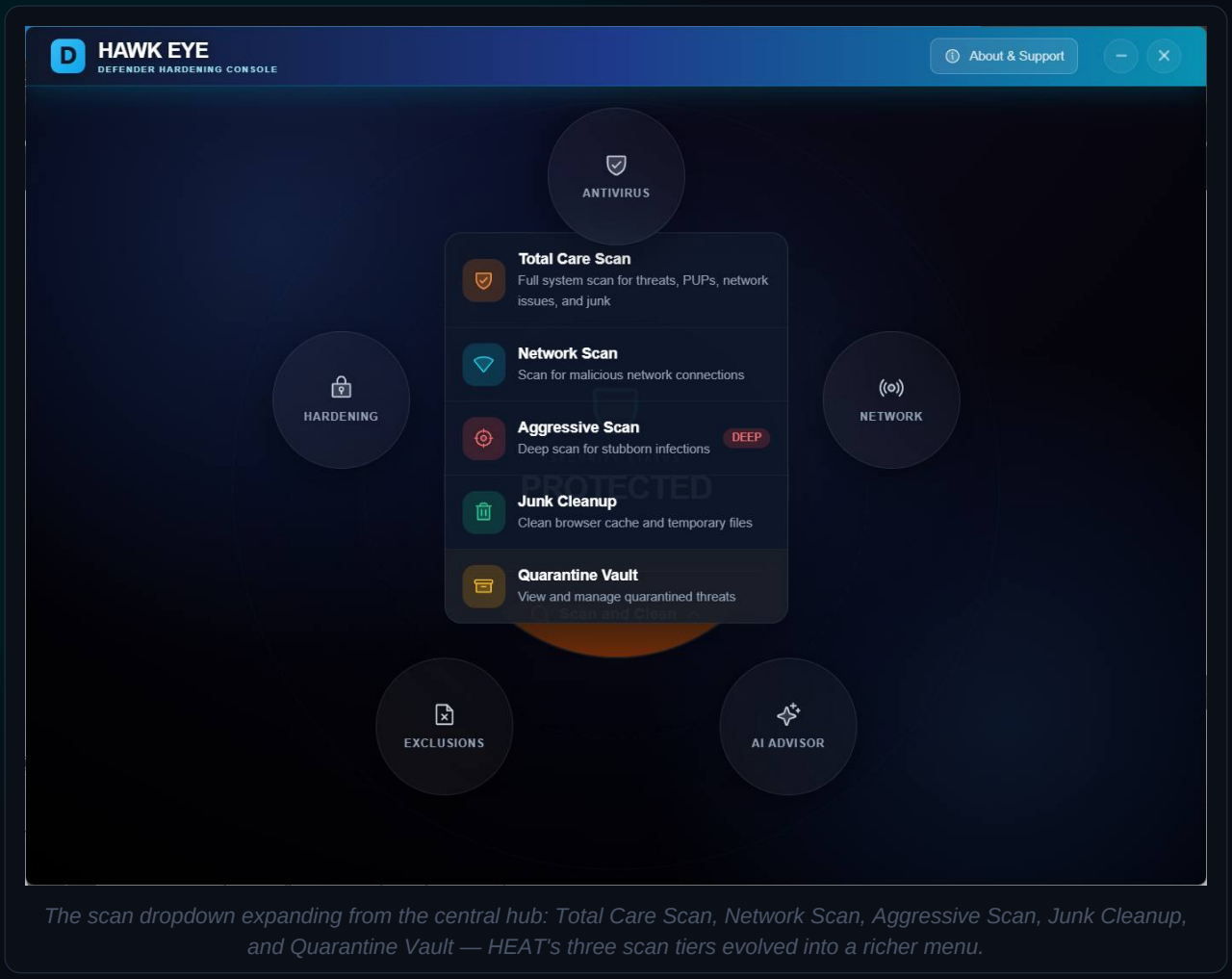
DHC's live scan interface: real-time progress ring at 76%, phase-specific 'Threat Detection' label, per-process enrichment status, and the Abort Scan option — all within the same application window.

The Radial Orbital Interface

The N360-inspired dashboard panels from HEAT were retired. In their place: a radial UI where the system's security posture sits at the centre and every capability — Network Protection, Hardening, Exclusions, AI Advisor — orbits around it, positioned with spatial hierarchy rather than sequential layout. The user doesn't read a dashboard top-to-bottom. They see the entire security posture at once, with the most important element at the centre.



The OrbitUI radial interface: Security Status at the centre, with Antivirus, Hardening, Network, Exclusions, and AI Advisor orbiting around it. The amber 'Scan and Clean' button anchors the bottom.



The scan dropdown expanding from the central hub: Total Care Scan, Network Scan, Aggressive Scan, Junk Cleanup, and Quarantine Vault — HEAT's three scan tiers evolved into a richer menu.

No other security product does this. Norton, Kaspersky, Bitdefender, Windows Defender — they're all sidebar-plus-list layouts. Every single one. The radial interface instantly signals that this isn't another antivirus dashboard wearing a dark theme. The glass morphism, the atmospheric particle effects, the ambient animations — all of it breathes better in a radial composition than in a rectangular grid. OrbitUI's name stopped being a metaphor and became the actual interaction model.

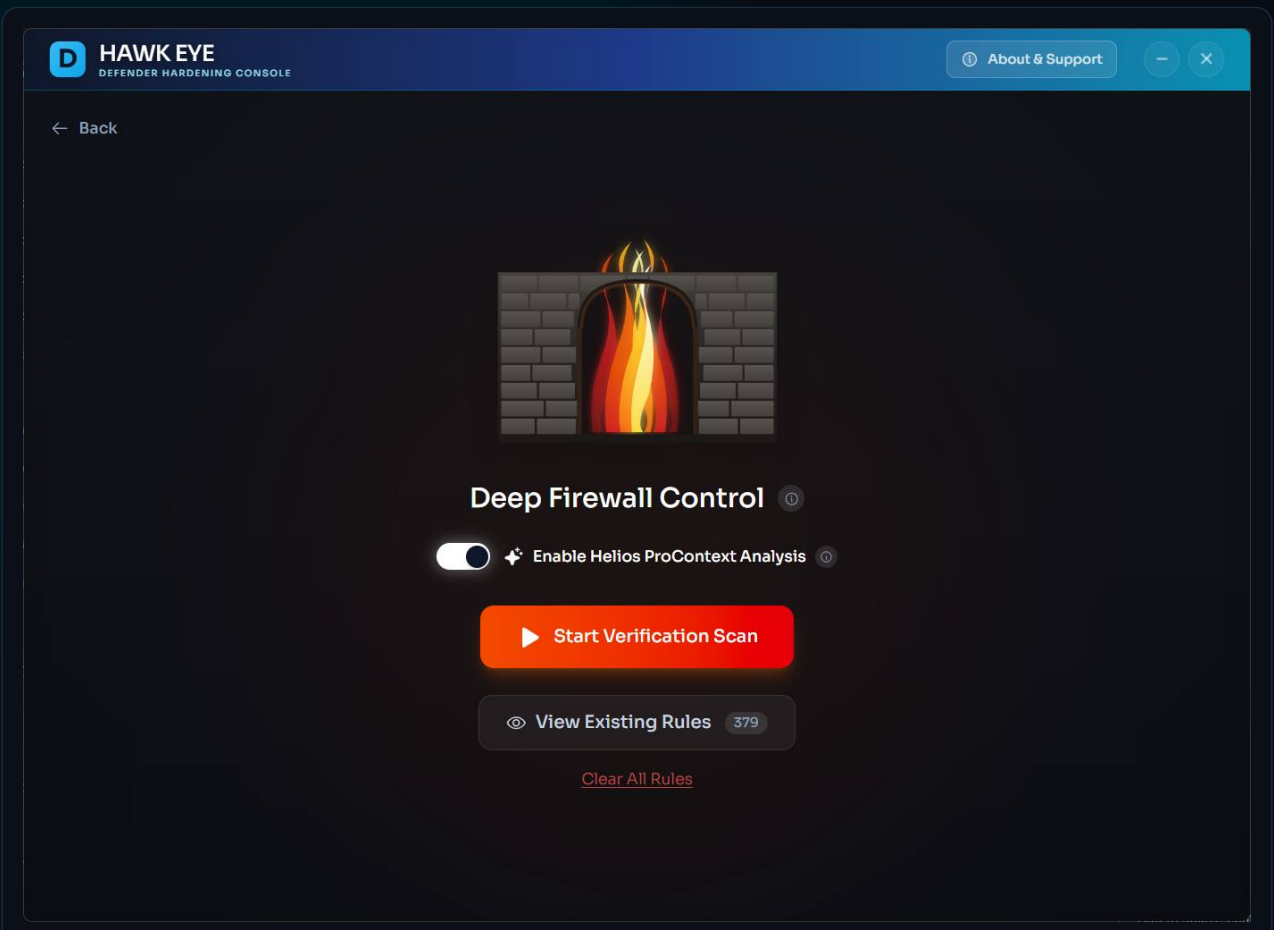
The N360 panels weren't removed. They were transcended.

The Platform Today

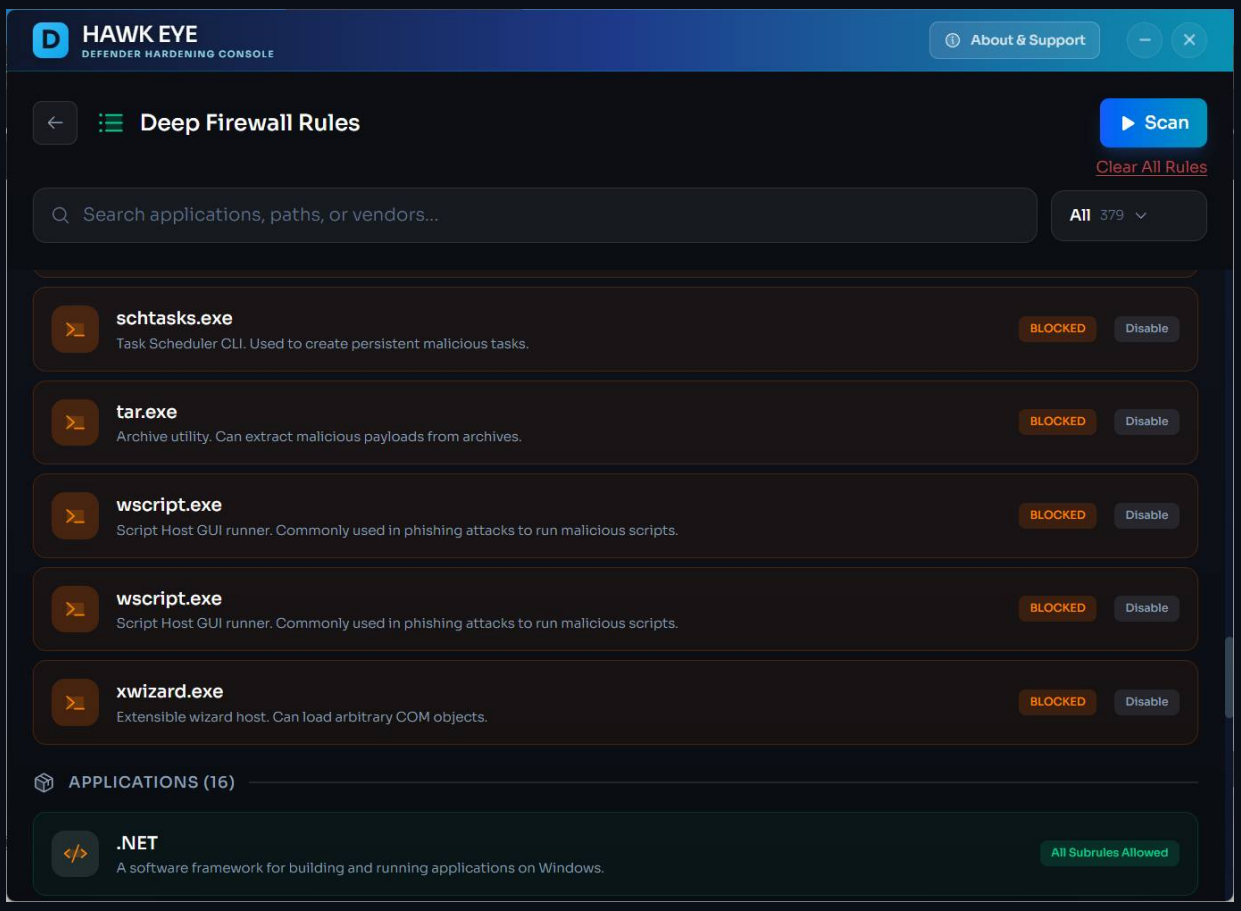
What began as a single PowerShell script is now a suite of security products unified under the Hawk Eye Analysis brand.

The **Defender Hardening Console** is the desktop hub — a C++23/React application with real-time threat monitoring and the OrbitUI design system. **Helios Web Marshall** extends protection to the browser through a Chrome/Firefox extension. **Talon** is an AI-powered shopping trust scanner. **The Inspector Hawk** is a scam analysis tool. **AiDefender** handles download analysis and triage.

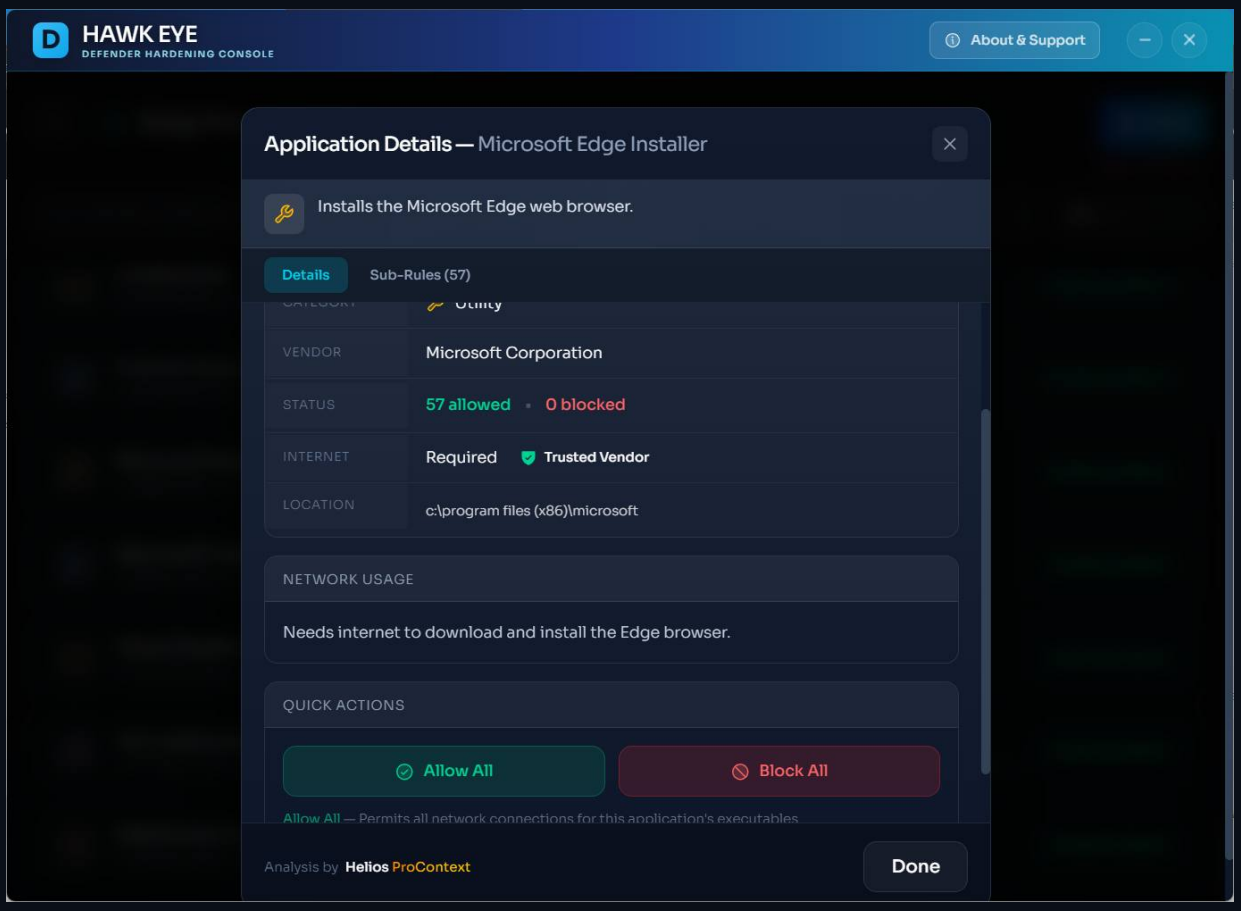
Each product has its own identity, but they share a common design language — OrbitUI's glass morphism, orbital animations, atmospheric effects, and amber-gold palette — and a common detection philosophy built on compound heuristic analysis.



Deep Firewall Control: the fireplace illustration, Helios ProContext Analysis toggle, and the verification scan — network protection evolved far beyond HEAT's connection monitor.



LOLBin blocking in action: schtasks.exe, wscript.exe, tar.exe, and xwizard.exe blocked at the firewall level — the same LOLBin awareness from HEAT's heuristics, now enforced proactively.



Helios ProContext application analysis: AI-powered descriptions, vendor trust verification, sub-rule counts, network usage context, and one-click Allow All / Block All actions.

What Survived, What Didn't, and What Mattered

The detection naming convention (`Helios:Category!Variant`) survived from the first Orion prototype to the current C++23 engine, unchanged. The threat intelligence knowledge base schema is the same structure today as it was in the original configuration file. The CPR Toolbox concept of proactive hardening actions became the philosophical backbone of a product literally named *Defender Hardening Console*.

The JSON-file-as-IPC pattern didn't survive, replaced by the WebView2 bridge. The 7-Zip quarantine dependency didn't survive, replaced by native file operations. The RunspacePool threading model didn't survive, replaced by `std::jthread` pools. Everything that was a workaround for PowerShell's limitations was replaced by the thing it was working around.

What mattered most was the four months of validation. By the time the C++23 rewrite began, every architectural question had already been answered. The threading model was proven. The detection heuristics were tested against real malware. The UX hierarchy was validated by real users. The product thinking was complete.

The PowerShell version wasn't a prototype that got thrown away. It was a proving ground that got promoted.

The Takeaway: Innovative Thinking Over Conventional Tools

If there's a recurring theme in this story, it's that the best solutions came from refusing to accept conventional constraints. PowerShell can't have a rich UI? Use the browser. WPF is the "correct" approach? Skip it — it's brittle and limited. Static IOC lists are how threat intelligence works? Build a system that derives its own threat patterns dynamically. Security tools should look utilitarian? Make it cinematic.

A conventional developer looks at constraints and works within them. Console menus, numbered prompts, maybe a Windows Forms dialog if they're feeling ambitious. They solve the problem as defined by the tools they have. The innovative approach asks a different question: not "what can my tools give me" but "what does the user need, and how do I get there regardless."

That instinct drove the browser-as-UI-framework invention. It drove the compound heuristic detection philosophy. It drove DynaTune. It drove the decision to write a full React SPA report for a PowerShell script. It drove the eventual leap to C++23 when the ideas outgrew their container. And it drives the product today — a radial orbital interface that no other security product has attempted, built by a company that started with a single script.

HEAT was an extraordinary piece of engineering for a PowerShell script. It was also, ultimately, a product too ambitious for its runtime. Both of those things can be true simultaneously, and the tension between them is what drove the evolution from script to platform.

The ideas were always bigger than the container. The container was what proved the ideas were worth building bigger.

Hawk Eye Analysis — Bring the heat on malware.