williballenthin via RSS

Reading List - 2026-01-01

Jan 1, 2026

Table of Contents

Build / Buy / Bot

Source: *https://sneak.berlin/20251224/build-buy-bot/*

In the old days when you ran your startup, you had to focus on your core objectives very closely in the early days. You had a limited number of innovation tokens → https://mattrickard.com/innovation-tokens, and beyond that you had to focus on your core product exclusively. No sysadminning (Heroku/Lambda), no hard drives (S3), no databases (RDS), no CDN (Cloudflare), no geth (Infura).

As a result, there are many SaaS startups that offer relatively straightforward services: things like APIs, API caching/aggregation/translation layers, industry-specific CRUD apps, image or video transcoding and resizing, analytics, webhook gateways → https://www.blocknative.com/blog/ethereum-transaction-notification-webhooks, etc. They're not hard to build, they're just time-consuming, and adjacent to your startup's core mission. You can pay them some small usage-based fee now, and if you get larger, you can build a replacement or renegotiate or change vendors.

A lot of SaaS startups' secret sauce was simply having faster devs than their customers and perhaps on-call sysadmins, not some specific insight or unique knowledge about the problem space they are solving.

That's all out the window now with SOTA AI models. Disparage "vibe coding" all you like, it's here and it works and it's going to change the world. → https://fly.io/blog/youre-all-nuts/

The build/buy equation just got an anvil dropped on one side of the seesaw, and your stupid SaaS startup idea is pulling a Wile E. Coyote midair at 30,000 feet. If all of the edge you had was that you could build it faster than your customers, you're toast.

Competent engineers can now crank out a prototype supporting api/service (unrelated to their startup's *raison d'être*) in a day or two, something that used to take a few people a month. Will it be as good and featureful and polished? No. Will it be as reliable? No, but serverless exists and bugfixes and test coverage will take minutes now instead of hours. It's a startup, and it'll probably be good enough, and close enough in perf to your free/startup tier that they'll do that instead of paying you $49/month.

Furthermore, they'll probably open source it, because it is unrelated to their profit model; remember, we're talking about tangential supporting services here. If they don't, someone else scratching that itch will. People will add features to it to make it more broadly useful on an as-needed basis. Bullshit "open core" startups will thankfully be toast.

The barrier to entry for starting a useful SaaS before was that you could actually ship working software faster than most. That's gone now.

I expect:

- many small orgs/startups choosing "build" much more often than "buy"
- large orgs assigning small teams to build internal replacements for medium-to-large SaaS products that they are paying a vendor big recurring dollars for, because of the force multipliers that are LLMs
- boring/basic CRUD b2b SaaSes to die even faster than they already are
- a slight shift back toward b2c due to the decreased cost of platform-specific mobile client development
- tons of SaaSes being replaced by f/oss projects
    - …many of which will sadly be fake open source → https://sneak.berlin/20250720/minio-are-assholes/
- "open core" projects to have their data retention/SSO/2FA enterprise carrot nonfree plugins/libraries to be either replaced by vibe coded f/oss replacements that slot in easily to the "community edition", or simply forked, as the cost of reviewing PRs and merging in upstream changes drops dramatically
- increased frequency of new working and useful f/oss projects, because scratching an itch becomes *so much easier and faster*
    - I even have a → https://git.eeqj.de/sneak/routewatch few → https://git.eeqj.de/sneak/mfer of → https://git.eeqj.de/sneak/ipapi my → https://git.eeqj.de/sneak/vaultik own → https://git.eeqj.de/sneak/secret. I imagine many others are tagging 1.0.0 of little side projects that have been pending for years.

Agents Done Right: A Framework Vision for 2026

Source: https://blog.bryanl.dev/posts/agent-framework-vision/

The Problem with Today's Agent Frameworks

I've been building with LLM-based agents for a while now, and I've been using a lot of other people's tools too. The space is growing fast, but something feels off. It's getting complex for the sake of being complex. New abstractions pile on top of old ones. Configuration options multiply. Steve Krug's → https://sensible.com/ first law of usability is "don't make me think." Current agent frameworks violate this constantly. And yet I keep running into the same walls.

The agent starts strong, but as the task gets complex, its context window fills up. The context window is the LLM's working memory, a limit on how much text it can hold at once, measured in tokens (roughly, chunks of words). Fill it up, and the model loses track. It forgets what it was doing. It repeats itself. It starts hallucinating, making things up with confidence. Eventually it fails. Not because the model isn't capable, but because the architecture around it can't manage the complexity.

We're in the "Ruby on Rails 1.0" era of agent development. Everyone is building agents, but we're all solving the same problems from scratch: context exhaustion, doom loops, model selection paralysis, and the cognitive overload of reviewing agent output. I once watched an agent run `npm run build` over and over for five minutes while I was on a phone call. It just kept going, burning tokens, stuck in a loop it couldn't escape. That's a doom loop. The agent repeats the same failed action, hoping for a different result.

The teams that have shipped production agents have independently discovered similar architectural patterns. But these insights are locked inside proprietary systems. The open source ecosystem is still dominated by thin wrappers around LLM APIs that avoid the hard problems.

It's 2026. We should have a framework that makes the *right* architecture the *easy* architecture. This post is my attempt to think through what that framework should look like.

Core Principles

1. Convention Over Configuration

Ruby on Rails succeeded because it made decisions for you. Database table names, file locations, URL structures were all derived from conventions. You could override them, but you didn't have to think about them.

Agent frameworks today are the opposite. Every project starts with: Which model? Which embedding provider? How should I structure tools? What's my context strategy? How do subagents communicate?

The framework should have strong defaults for all of this. Here are the conventions I'd choose:

Model selection by task complexity. Simple edits (adding a log statement, fixing a typo) use a fast model. Multi-file refactors or debugging sessions use a reasoning model. The framework infers complexity from the task description and codebase scope. You don't specify a model.

Context budgets with inheritance. Each agent gets a context budget. Subagents inherit a portion of their parent's remaining budget. When an agent approaches its limit, the framework triggers automatic summarization. This forces the architecture toward delegation: if a task doesn't fit in your budget, spawn a subagent.

Mandatory checkpoints for risky operations. File deletion, multi-file modifications, and detected uncertainty (phrases like "I'm not sure" or "this might break") require human approval. Everything else proceeds automatically. You opt out of safety, not into it.

Curated tool profiles by archetype. Searchers get search tools. Writers get write tools. Researchers get web access. Archetypes don't get tools outside their profile unless explicitly granted. This prevents the "too many tools" problem where agents waste context evaluating irrelevant options.

These conventions encode lessons from watching agents fail. Context exhaustion, review fatigue, tool confusion: the failure modes are predictable. The conventions exist to make the right architecture the easy architecture.

```
// This should just work - all conventions applied
const agent = new Agent({ codebase: "./my-project" });
await agent.run("Add input validation to the signup form");
```

Override when you need to. But start productive.

---

2. Tasks, Not Models

The current generation of tools asks: “Which model do you want to use?”

This is the wrong question. For most tasks, users shouldn’t be thinking about models at all. When you ask an agent to “add input validation to the signup form,” you don’t care whether it uses a fast model or a reasoning model. You care that it works.

Model selection is an optimization detail. You describe what you want; the framework figures out how to deliver it efficiently. A quick code edit optimizes for speed. Complex debugging allocates thinking time. Large refactors balance cost and capability. You shouldn’t have to care which model runs under the hood.

Framework implication: No `model` parameter in the default API. The framework infers requirements from the task. Power users can override when needed, but the default path requires zero model knowledge.

---

3. Subagents as the Primary Scaling Mechanism

Watch an agent tackle a complex task. It reads files, searches the codebase, backtracks, tries a different approach. The context window fills up. By the time it’s ready to write code, it’s forgotten the original requirements. It loops. It hallucinates. It fails.

I hit this wall constantly when researching new topics. One PDF can consume your entire context window. A large codebase is even worse. You end up working in awkward, inconvenient ways just to get access to the knowledge you need.

An obvious fix is to “make context windows bigger.” But bigger windows just delay the problem. They’re slower and more expensive.

A better approach is *factoring*: break the work into isolated subtasks, each with its own context. A subagent searches the codebase and returns only the relevant file paths. Another analyzes a specific function and returns only its findings. The parent agent stays focused, receiving distilled results instead of accumulating everything.

Subagents are to agents what functions are to programs. They isolate context for subtasks and return only relevant results to the parent. They can be optimized for different task types, whether speed or depth, and they enable parallelism without context collision.

The difference is dramatic. A single-agent approach might use 90% of its context window stumbling through a task. The same task with subagents? The parent uses 25%, staying sharp throughout.

Spawning subagents should feel as natural as calling a function. Not an advanced feature you learn later, but the default pattern from day one.

Framework implication:

```
// This should feel as natural as a function call
const result = await agent.delegate("searcher", {
  task: "locate authentication logic",
  returnFormat: "file_paths_with_snippets"
});
```

The framework also needs primitives that make context management automatic:

```
const agent = new Agent({
  contextBudget: 128_000,          // Explicit budget, inherited by subagents
  overflowStrategy: "summarize",    // What to do when approaching limit
  loopDetection: true              // Circuit-break on repetitive patterns
});
```

With these primitives, agents can focus on the task while the framework handles the resource management. Just like garbage collection lets programmers focus on logic instead of memory.

---

4. Opinionated Subagent Archetypes

Watch enough production agents and you'll see the same patterns emerge. Teams independently discover that certain subtask shapes keep recurring, and that specialized subagents for those shapes dramatically outperform general-purpose ones.

Here are the archetypes that have proven themselves:

Archetype	What it does	Why it's specialized
Searcher	Finds relevant code, files, symbols	Optimized for speed; returns locations, not content
Thinker	Reasons through complex problems	Allocates thinking time; returns analysis, not action
Researcher	Gathers external knowledge (docs, APIs)	Web/retrieval access; returns synthesized context
Writer	Makes targeted code changes	Diff-focused; returns patches, not explanations
Planner	Breaks down complex tasks	Strategic focus; returns task breakdown
Checker	Validates and critiques work	Independent perspective; adversarial stance

Notice the pattern: each archetype has a *constrained output format*. A Searcher returns locations. A Thinker returns analysis. A Writer returns patches. This constraint is the key. It forces the subagent to distill its work rather than dump everything into the parent's context.

These archetypes should ship as built-in, customizable components. You shouldn't have to reinvent the Searcher pattern from scratch. It should just be there, ready to use, with sensible defaults you can override when needed.

---

5. Human-Agent Workflow is Part of the Framework

The bottleneck has shifted. It's no longer "can the agent write code?" It's "can the human review and trust the code fast enough?"

Today's workflow: agent dumps a wall of changes, human squints at diffs, tries to understand the intent, hopes nothing broke. This doesn't scale. As agents get more capable, the review burden grows faster than human attention.

When you run an agent, you should be able to specify where you want control. Checkpoints give you natural stopping points for review. Change proposals explain what will change and why before changes are made. Validation hooks let you plug in secondary verification (another model, static analysis, or tests). Diff streaming gives real-time visibility into changes as they happen. And approval gates let you require sign-off for high-risk operations.

Here's what that looks like:

```
const result = await agent.run({
  task: "Add rate limiting to /api/users",
  checkpoints: ["before_write", "after_plan"],
  requireApproval: ["delete_file", "modify_config"],
  onProposal: (plan) => showPlanToUser(plan)
});
```

This isn't UI. It's protocol. The framework defines the *contract* between agent and human; UIs implement it. A CLI might show a simple approve/reject prompt. An IDE might render an interactive diff viewer. Same protocol, different presentations.

---

6. Tools are Curated, Not Collected

The promise of universal tool interoperability sounds great in theory. In practice: more tools = more confusion.

Here's why. Every tool you give an agent is a decision it has to make: "Should I use this?" With 5 well-chosen tools, that decision is easy. With 50 tools, many overlapping, some poorly documented, others rarely useful, the agent wastes context evaluating options, makes false starts with wrong tools, and loses focus on the actual task.

I ran into this recently with MCP (Model Context Protocol) → https://modelcontextprotocol.io/ servers. I built one for Outlook to read my mail and calendar. But with multiple MCP servers installed, the agent kept trying to look up information on the web instead of using the mail tool sitting right there. Too many options, not enough guidance about which one to pick.

What you want is a curated core toolset optimized for coding tasks, not a grab-bag of integrations. Different subagents should get different tool subsets: a Searcher doesn't need web access. Destructive tools should require human approval. And tool usage patterns should be tracked to surface which tools help vs. which cause thrashing.

Framework implication:

```
const searcher = new Searcher({
  tools: ["grep", "ast_search", "file_read"],  // Curated subset
  restricted: ["web_fetch", "file_write"]      // Explicitly excluded
});

// Tools are first-class objects with metadata
const grepTool = {
  name: "grep",
  description: "Search file contents with regex",
  whenToUse: "Looking for specific strings or patterns in code",
  costEstimate: "low",
  requiresApproval: false
};
```

The goal: agents that pick the right tool immediately, not agents that waste time evaluating irrelevant options.

---

7. When to Use a Subagent vs. a Tool

There's a simple heuristic for deciding whether something should be a tool or a subagent:

Tool: Stateless transformation. Input goes in, output comes out, no iteration required. Format a date. Parse JSON. Run a regex. The operation doesn't accumulate context or require judgment.

Subagent: Iteration, judgment, or context accumulation. The operation explores, backtracks, tries alternatives, or builds up intermediate state. Search, research, analysis, planning: these need their own context window.

The architectural reason to care: if you implement an iterative operation as a tool, its entire execution trace pollutes the parent context. Every search result, every intermediate step, every dead end. All of it crowds out space for the actual task.

```
// Bad: "tool" that leaks context
const searchTool = {
  name: "search_codebase",
  execute: async (query) => {
    // All of this ends up in parent context
    const files = await grep(query);
    const ranked = await rerankResults(files);
    const snippets = await extractSnippets(ranked);
    return snippets;
  }
};

// Good: subagent with isolated context
const searcher = new Searcher({
  // Runs in own context window
  // Only `snippets` returned to parent
  returns: "snippets_with_locations"
});
```

This matters because many things we *call* tools are actually subagents in disguise. Web search, documentation lookup, code analysis: these involve iteration and judgment. The name “tool” makes them sound simple, but they’re not. Implementing them as proper subagents with isolated context is what makes the archetype pattern (Searcher, Thinker, Researcher) work.

---

8. Subagent Context is Ephemeral by Default

A subagent’s working context is like local variables in a function. It exists during execution and is discarded when done.

Consider a Researcher subagent investigating a library. It fetches 10 documentation pages, reads API references and examples, follows links to GitHub issues, and synthesizes findings into a summary. All of that is working context. But it returns only the summary to the parent.

The parent doesn’t need the 10 pages, the API refs, or the GitHub issues. It needs the answer. All that intermediate work is temporary scaffolding.

```
const researcher = new Researcher({
  // Everything inside runs in ephemeral context
  // Only the return value persists
  returns: "synthesis",

  // Optional: persist specific artifacts
  persist: ["key_code_snippets", "api_signatures"]
});

// Parent receives ~500 tokens, not 50,000
const findings = await agent.delegate(researcher, {
  task: "How does auth work in this library?"
});
```

The principle: Subagents accumulate context to do their job, then *compress* before returning. The parent receives a distilled result, not the full execution trace.

I built something like this to explore a large codebase. A custom agent walked through the source files, read them, and wrote summarized markdown documentation. The agent consumed thousands of lines of code, but what I got back was a concise capture of the service's flow. The essence, not the exhaustive detail. That's the pattern: do the heavy lifting in isolated context, return only what matters.

This is how human experts work too. When you ask a colleague to research something, you want their conclusion, not everything they read along the way.

The Developer Experience

What Building an Agent Should Feel Like

All of the principles above collapse into a simple question: what does it feel like to build with this framework? If the conventions are right, the code should be obvious. You declare what you want, not how to manage it. The framework handles orchestration, context, and human checkpoints. You focus on the task.

Here's what that looks like:

```
import { Agent, Searcher, Thinker, Writer } from "agentkit";

// Declare a coding agent with specialized subagents
const agent = new Agent({
  name: "code-assistant",
  mode: "smart", // vs "fast" for quick iteration
  subagents: [
    new Searcher({ tools: ["grep", "ast_search", "embeddings"] }),
    new Thinker({ timeout: 120 }),
    new Writer({ requireApproval: false }),
  ],
  contextBudget: 128_000,
  humanCheckpoints: ["before_multi_file_edit", "on_uncertainty"],
});

// Run a task - framework handles subagent orchestration
const result = await agent.run({
  task: "Add rate limiting to the /api/users endpoint",
  codebase: "/path/to/repo",
});

// Result includes structured output for UI integration
result.changes;      // List of file changes
result.proposal;     // What changed and why
result.contextUsed;  // Debugging/optimization info
```

Why TypeScript?

TypeScript offers patterns that make agent code safer in ways that Python's type system can't match.

Branded types for resource budgets. Context tokens aren't just numbers. They're a distinct unit. Branded types prevent you from accidentally passing a line count where a token count is expected:

```
type ContextTokens = number & { readonly brand: unique symbol };
const budget: ContextTokens = 64_000 as ContextTokens;
// Can't accidentally pass a plain number where ContextTokens is required
```

Discriminated unions for checkpoint states. When a checkpoint can be pending, approved, or rejected, discriminated unions force exhaustive handling. The compiler catches missing cases at build time, not runtime:

```
type CheckpointState =
  | { status: "pending" }
  | { status: "approved"; by: string; at: Date }
  | { status: "rejected"; reason: string };

function handleCheckpoint(state: CheckpointState) {
  switch (state.status) {
    case "pending": return showWaiting();
    case "approved": return proceed(state.by);
    case "rejected": return showError(state.reason);
    // TypeScript errors if you miss a case
  }
}
```

Generic constraints for subagent return types. A `Searcher<FileLocation[]>` and a `Thinker<Analysis>` are different types. The parent agent knows exactly what shape to expect from each delegation:

```
const locations = await agent.delegate<FileLocation[]>(searcher, { task: "find auth"
});
// locations is typed as FileLocation[], not unknown
```

Beyond type safety, the ecosystem fits: VS Code extensions, language servers, and most developer tooling already run on TypeScript. Using Bun → https://bun.sh/, agents compile to standalone executables with no runtime dependencies.

What This Enables

Think about what web development looked like before Rails. Every project started with the same decisions: How do I structure my code? How do I talk to the database? How do I handle routing? Teams spent months on plumbing before writing a single line of business logic. Rails changed that by encoding the answers into conventions. Suddenly, developers could go from idea to working application in hours instead of weeks.

Agent development is in that pre-Rails moment right now. Every team building production agents is solving the same problems: context management, subagent orchestration, human review workflows, tool selection. The solutions exist, but they're locked inside proprietary systems or tribal knowledge.

With the right framework primitives, agent builders can focus on what makes their agent unique instead of reinventing context management for the hundredth time. They can experiment with novel interaction patterns instead of debugging the same doom loops. They can invest in evaluation and domain expertise instead of infrastructure.

And users get agents that actually work. Not agents that succeed on simple tasks and fall apart on complex ones. Not agents that require babysitting to avoid going off the rails. Agents with predictable behavior, transparent operation, and graceful degradation when things get hard.

Open Questions

There's still work to figure out. These are the problems I'm thinking through, along with my current hunches.

How do subagents share learned context? There's probably a "memory" layer that persists across tasks. Something like a project-level knowledge base that subagents can read from and write to. The Researcher finds something important, it goes into shared memory. The Writer pulls from it later. This doesn't have to be LLM-backed memory. It could be a graph database, a structured knowledge store, or something we haven't invented yet. But the interface has to be simple enough that it doesn't become another configuration burden.

How do peer agents collaborate? Most agent architectures are hierarchical. Parent spawns child, child returns result. But some tasks need peers working in parallel, sharing findings as they go. I suspect the answer is message-passing with typed channels, similar to how concurrent systems handle coordination. The framework handles the plumbing; agents just send and receive.

How do you evaluate agents systematically? This might be the hardest problem, and I don't think anyone has solved it yet.

The core challenge is ground truth. To evaluate whether an agent did the right thing, you need to know what "right" means for that task. For coding tasks, you can check some things automatically (does the code compile? do tests pass?) but these only catch obvious failures. An agent can produce working code that's architecturally wrong, or secure code that's unmaintainable.

I think evaluation has to happen at multiple layers. **Task success** is the baseline: did the change work at all? **Behavioral consistency** asks whether the same task on the same codebase produces similar results across runs. If an agent gives wildly different answers each time, something's wrong. **Human preference** captures whether humans actually accept the changes in practice.

The infrastructure for this is expensive to build. You need captured scenarios (codebase + task + expected behavior), replay with controlled randomness, and scoring that's more nuanced than pass/fail. It's similar to how you'd evaluate a human candidate through realistic work samples rather than abstract tests. The framework should probably include hooks for scenario capture and replay, even if the evaluation logic is user-defined.

There's also a meta-problem: evaluating the evaluations. A scenario that tests whether the agent can add two numbers tells you nothing useful. The scenarios themselves need to be validated for difficulty and discriminative power. Do they distinguish good agents from bad ones? Do they catch regressions? This is the same challenge test engineers face with code coverage: 100% coverage means nothing if the tests are trivial.

My intuition is that evaluation requires three primitives: **Scenario** (a frozen codebase + task), **Assertion** (did the output compile, pass tests, match intent), and **Consistency** (same scenario, N runs, what's the variance). This is closer to integration testing than unit testing.

How do you budget spend across subagent hierarchies? Context tokens are one cost, but API calls add up fast when you're spawning subagents. The framework probably needs a cost model that propagates budgets down the hierarchy. Parent allocates a budget, subagents inherit portions of it, and the framework enforces limits before you get a surprise bill.

What happens when a subagent fails? The boring answer is probably the right one: retry with exponential backoff, then escalate to the parent with an error. The parent decides whether to substitute a different approach or surface the failure to the user. Automatic substitution sounds clever but might hide problems that humans should see.

What's Next

The patterns are emerging. Production agent systems have proven what works. Now we need to turn these lessons into infrastructure that benefits everyone.

I started this post frustrated with complexity. Every agent framework I tried added abstraction layers without solving the hard problems. Configuration options multiplied while agents still choked on context. New features shipped while doom loops went unfixed. I kept trying new tools and spending more time learning their paradigms than actually getting work done. That's a red flag. It means we haven't found the right way to look at this space yet.

Ruby on Rails became popular not because it introduced a lot of new ideas, but because it made easy things trivial and hard things possible in a reasonable amount of time. That's why everyone learned Ruby. We need the same thing for agents.

The answer isn't more complexity. It's the right complexity: conventions that encode hard-won lessons, primitives that make the correct architecture easy, and defaults that work out of the box.

I don't think there's one framework to rule them all. But the patterns matter. Whether you're building agents, evaluating frameworks, or just trying to understand why your agent keeps running `npm run build` in a loop, these architectural ideas can help you reason about what's going wrong and what to try next.

If more people internalized these patterns, we'd all waste less time on plumbing and more time on problems that actually matter. And that's the point.

Hacker Book — Community, All the HN Belong to You! 2006

Source: https://hackerbook.dosaygo.com/

Viewing HN on Someday, Month 00, 0000. Times are relative to 11:59 PM.
Made by DOSAYGO → https://dosaygo.com/ · [GET THIS] → https://dosaygo-studio.github.io/HackerBook/

Replacing JS with just HTML - HTMHell

Source: https://www.htmhell.dev/adventcalendar/2025/27/

by Aaron T. Grogg → https://aarontgrogg.com/ published on Dec 27, 2025

For many years now, JavaScript has been the workhorse of the web. If you wanted to do something that couldn't be done with just HTML and CSS, you could usually find a way to do it with JS.
And that is great! JS has helped push user experiences forward, and honestly helped push HTML and CSS forward!

But as time marches on, and the HTML and CSS methods gain traction → https://webstatus.dev/?q=%28group%3Ahtml+OR+css%29&sort=name_asc, we need to start replacing the old JS methods that feel so comfy with new methods that require less JS.

Nothing against JS, but it has better things to do than setup and manage your accordions or offscreen navigation menus... Plus, JS needs to be downloaded, decompressed, evaluated, processed, and then often consumes memory to monitor and maintain features. If we can *hand-off* any JS functionality to native HTML or CSS, then users can download less stuff, and the remaining JS can pay attention to more important tasks that HTML and CSS can't handle (yet).

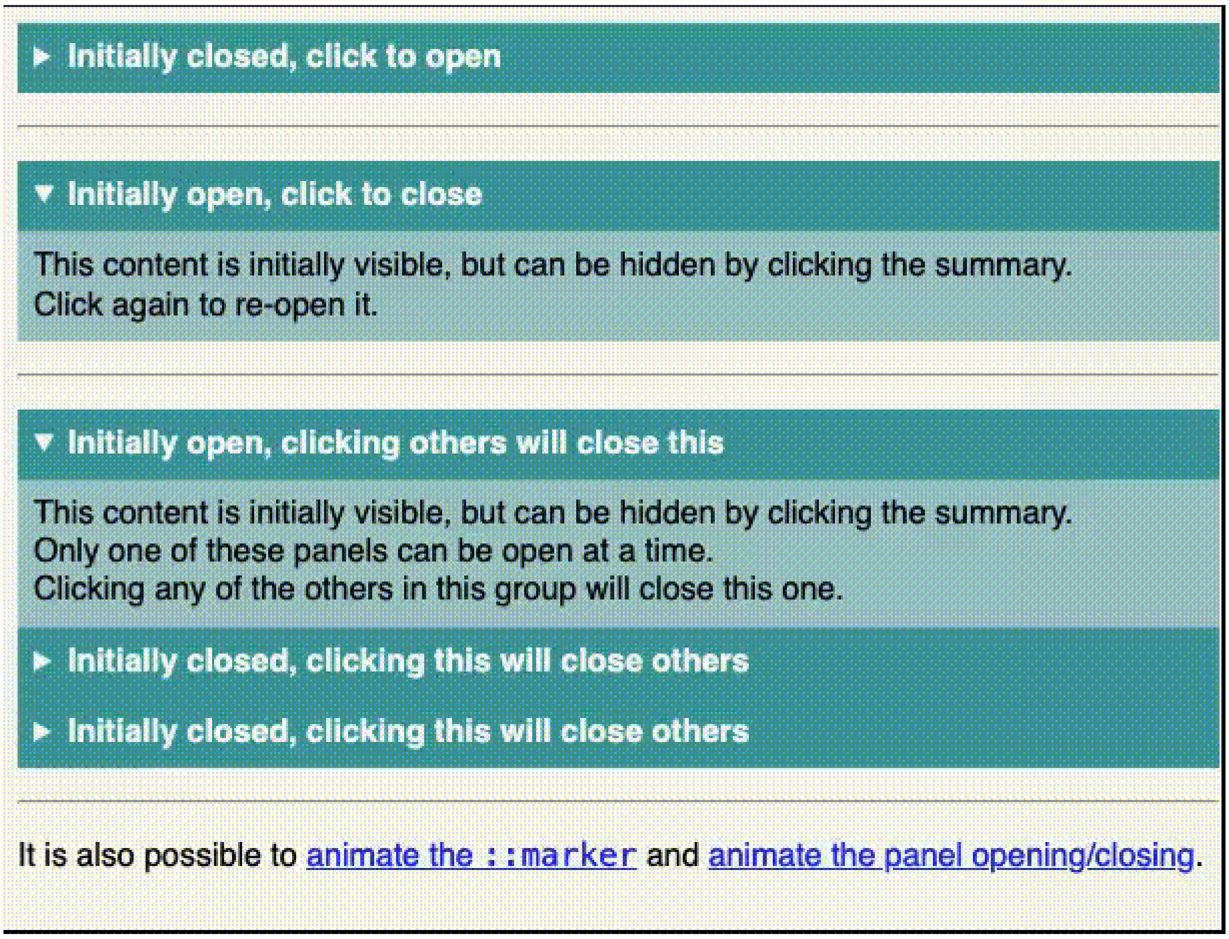
Below are a few examples; any you care to add?

Table of Contents:

Accordions / Expanding Content Panels

Description:

The `details` and `summary` HTML elements provide an HTML-only replacement to the typical JS accordion:


▸ Initially closed, click to open
▾ Initially open, click to close

This content is initially visible, but can be hidden by clicking the summary.
Click again to re-open it.

▾ Initially open, clicking others will close this

This content is initially visible, but can be hidden by clicking the summary.
Only one of these panels can be open at a time.
Clicking any of the others in this group will close this one.

▸ Initially closed, clicking this will close others
▸ Initially closed, clicking this will close others

It is also possible to animate the `::marker` and animate the panel opening/closing.

CopePen: Accordion / Expanding Content → https://codepen.io/aarontgrogg/pen/GgoOqVX

Use cases:

- Hiding/showing content
- Expanding content sections

Basic implementation:

```
<details>
  <summary>Initially closed, click to open</summary>
  Content is initially hidden, but can be revealed by clicking the summary.
</details>
```

Add an open attribute to set the default appearance as "open":

```
<details open>
  <summary>Initially open, click to close</summary>
  Content is initially visible, but can be hidden by clicking the summary.
</details>
```

Use the same name attribute on all related `details` (like radio buttons) to restrict only one open panel at a time:

```
<details name="foo" open>
  <summary>Initially open, clicking others will close this</summary>
  Content is initially visible, but can be hidden by clicking the summary; only one
panel can be open at a time.
</details>
<details name="foo">
  <summary>Initially closed, clicking will open this, and close others</summary>
  Content is initially hidden, but can be revealed by clicking the summary; only one
panel can be open at a time.
</details>
<details name="foo">
  <summary>Initially closed, clicking will open this, and close others</summary>
  Content is initially hidden, but can be revealed by clicking the summary; only one
panel can be open at a time.
</details>
```

You can also customize the appearance with CSS and trigger the open/close via JS.

Learn more about the `details` element in the previously-published "For the Love of <details> → https://www.htmhell.dev/adventcalendar/2025/23".

Resources:

- MDN `details` page → https://developer.mozilla.org/en-US/docs/Web/HTML/Element/details
- <details> element → https://www.codewithshripal.com/playground/html/details-element

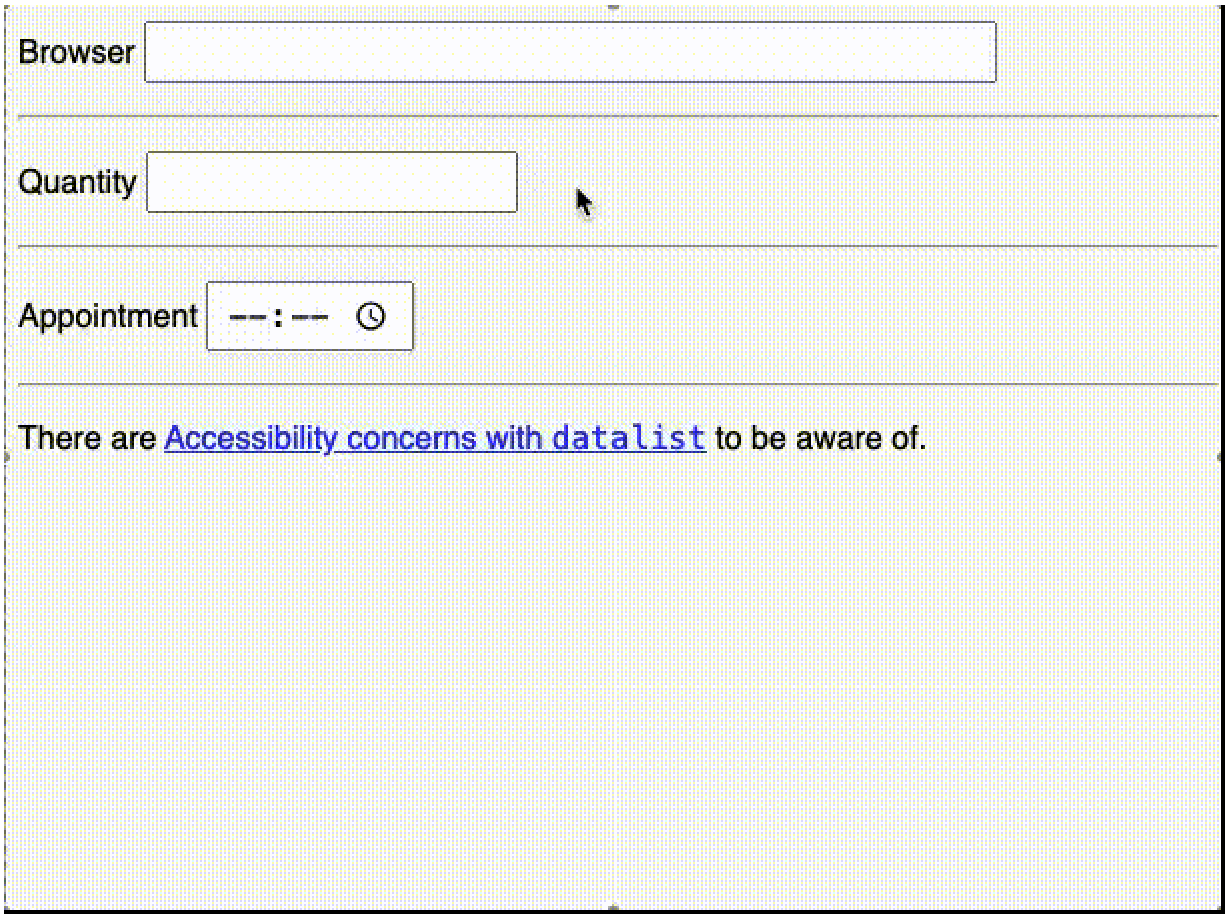
Browser compatibility:

- MDN `details` Browser Compatibility → https://developer.mozilla.org/en-US/docs/Web/HTML/Element/details#browser_compatibility

Input with Autofilter Suggestions Dropdown

Description:

Combining the HTML `input` and `datalist` elements can create a dropdown of options that autofilters as you type:



Browser

Quantity

Appointment --:--

There are Accessibility concerns with `datalist` to be aware of.

CodePen: Input with Autofilter Suggestions Dropdown → https://codepen.io/aarontgrogg/pen/yyePPor

Use cases:

- Site search
- Product search or filter
- Filter any list of data

Basic implementation:

```
<label for="browser">Browser</label>
<input type="text"
       list="browsers"
       id="browser" name="browser"
       size="50"
       autocomplete="off" />
<datalist id="browsers">
  <option value="Arc"></option>
  <option value="Brave"></option>
  <option value="Chrome"></option>
  <option value="DuckDuckGo"></option>
  <option value="Firefox"></option>
  <option value="Microsoft Edge"></option>
  <option value="Opera"></option>
  <option value="Safari"></option>
  <option value="Tor"></option>
  <option value="Vivaldi"></option></option>
</datalist>
```

You can also use other input types:

```
<label for="quantity">Quantity</label>
<input type="number"
       list="quantity-options"
       id="quantity" name="quantity" />
<datalist id="quantity-options">
  <option value="1"></option>
  <option value="2"></option>
  <option value="5"></option>
  <option value="10"></option>
  <option value="20"></option>
  <option value="50"></option>
</datalist>

<label for="appointment">Appointment</label>
<input type="time"
       list="appointments"
       id="appointment" name="appointment" />
<datalist id="appointments">
  <option value="12:00"></option>
  <option value="13:00"></option>
  <option value="14:00"></option>
</datalist>
```

Note that, at the time of this writing, Firefox was limited to only textual-based input types, so no `date`, `time`, `range` or `color` for now... :-(

Also note that, at the time of this writing, there are limitations on mobile, and accessibility concerns → https://htmhell.dev/adventcalendar/2025/5/#the-list-attribute-on-lessinputgreater-elements.

Resources:

- MDN `datalist` page → https://developer.mozilla.org/en-US/docs/Web/HTML/Element/datalist
- HTML5 datalist autocomplete → https://www.sitepoint.com/html5-datalist-autocomplete/

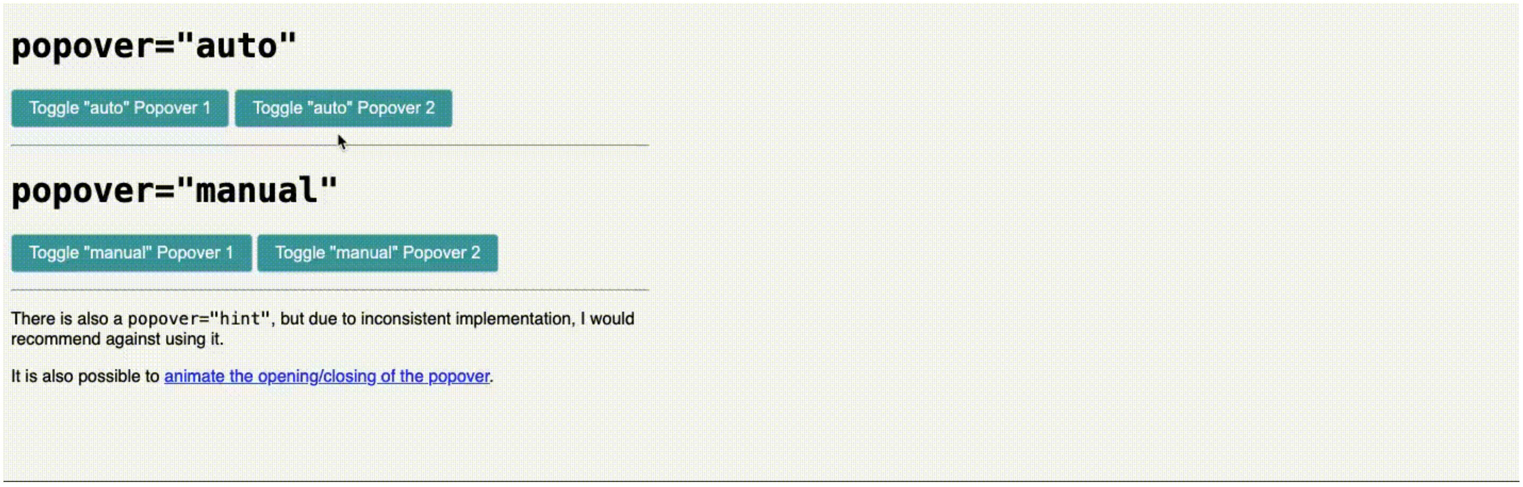
Browser compatibility:

- MDN `datalist` Browser Compatibility → https://developer.mozilla.org/en-US/docs/Web/HTML/Element/datalist#browser_compatibility

Modals / Popovers

Description:

The `popover` and `popovertarget` attributes can replace the traditional JS-driven modal/popover/overlay:



popover="auto"

popover="manual"

There is also a `popover="hint"`, but due to inconsistent implementation, I would recommend against using it.

It is also possible to animate the opening/closing of the popover.

CodePen: Modal / Popover → https://codepen.io/aarontgrogg/pen/QwyOKNW

Use cases:

- Hiding/showing side panels / additional information

Basic implementation

An `auto` popover (default) can be "light dismissed" (clicking outside of it or hitting the `esc` key). Opening an `auto` automatically closes any other `auto` popovers that were open. Clicking the `button` a second time will close the one it opened.

```
<button popovertarget="pop-auto">
   Toggle Popover
</button>
<dialog popover id="pop-auto">
   I'm an "auto" Popover!
</dialog>
```

A `hint` popover can also be "light dismissed". It does *not* close other `hint` popovers when opened. Clicking the `button` a second time will close the one it opened.

```
<button popovertarget="pop-hint">
   Toggle Popover
</button>
<dialog popover="hint" id="pop-hint">
   I'm a "hint" Popover!
</dialog>
```

Note that, at the time of this writing, Firefox and all iOS varieties do not support `hint` popovers.

A `manual` popover can *not* be "light dismissed". It does *not* close other `manual` popovers when opened. Clicking the `button` a second time will close the one it opened.

```
<button popovertarget="pop-manual">
   Toggle Popover
</button>
<dialog popover="manual" id="pop-manual">
   I'm a "manual" Popover!
</dialog>
```

Learn more about the opening and closing dialogs and popovers in the previously-published "Controlling dialogs and popovers with the Invoker Commands API → https://www.htmhell.dev/adventcalendar/2025/7".

Resources:

- MDN `popover` page → https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/popover
- Popover API examples → https://mdn.github.io/dom-examples/popover-api/
- Introducing the popover API → https://developer.chrome.com/blog/introducing-popover-api/
- Popover API! → https://codepen.io/jh3y/pen/XWPBmmo
- HTMHell #5 - An introduction to the popover attribute → https://www.youtube.com/watch?v=KX8YQW7stzs
- The accessibility of the popover attribute → https://www.youtube.com/watch?v=RVxl5nZY790

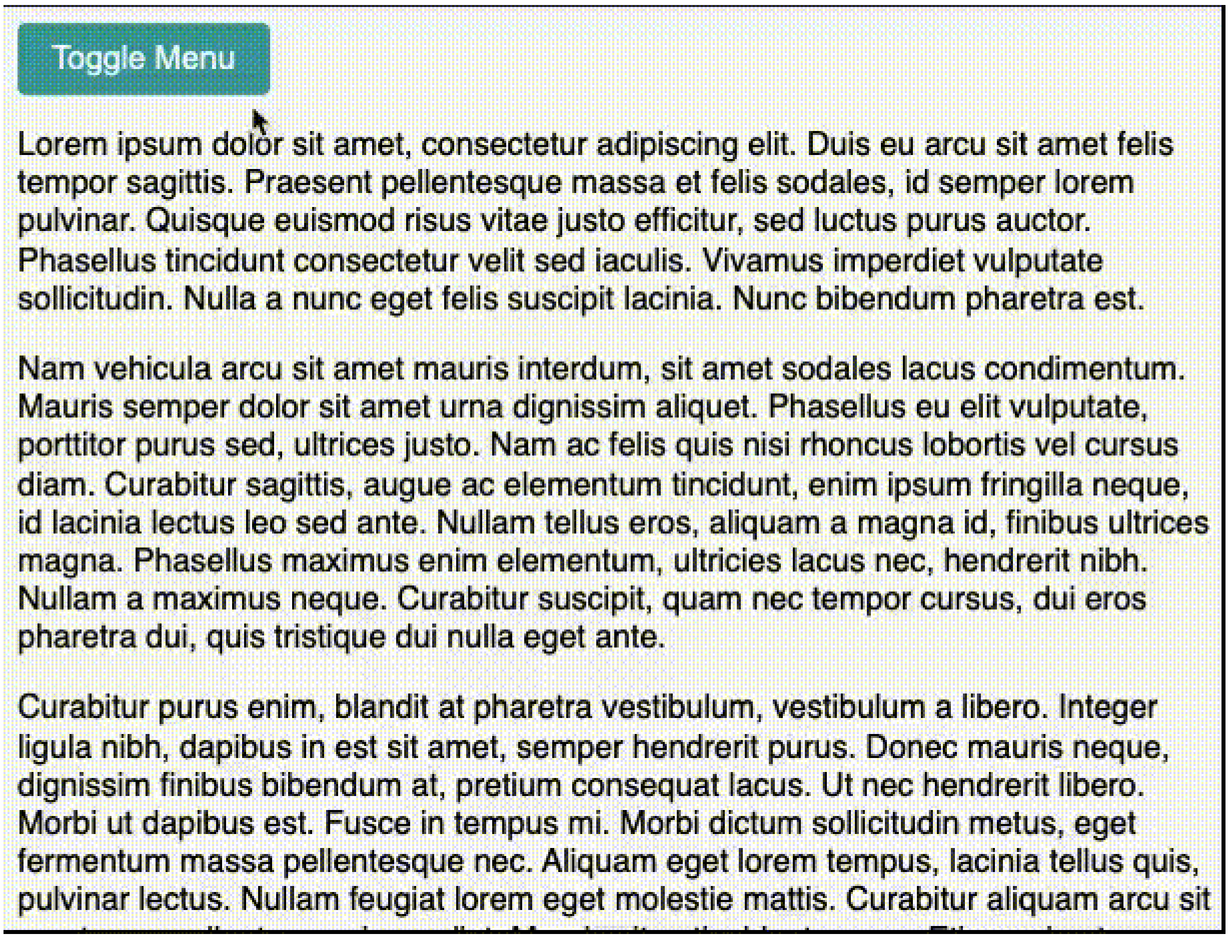
Browser compatibility:

- MDN `popover` Browser Compatibility → https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Global_attributes/popover#browser_compatibility

Offscreen Nav / Content

Description:

The above Modal / Popover functionality can also be used to create an offscreen navigation that requires no JS:



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Duis eu arcu sit amet felis tempor sagittis. Praesent pellentesque massa et felis sodales, id semper lorem pulvinar. Quisque euismod risus vitae justo efficitur, sed luctus purus auctor. Phasellus tincidunt consectetur velit sed iaculis. Vivamus imperdiet vulputate sollicitudin. Nulla a nunc eget felis suscipit lacinia. Nunc bibendum pharetra est.

Nam vehicula arcu sit amet mauris interdum, sit amet sodales lacus condimentum. Mauris semper dolor sit amet urna dignissim aliquet. Phasellus eu elit vulputate, porttitor purus sed, ultrices justo. Nam ac felis quis nisi rhoncus lobortis vel cursus diam. Curabitur sagittis, augue ac elementum tincidunt, enim ipsum fringilla neque, id lacinia lectus leo sed ante. Nullam tellus eros, aliquam a magna id, finibus ultrices magna. Phasellus maximus enim elementum, ultricies lacus nec, hendrerit nibh. Nullam a maximus neque. Curabitur suscipit, quam nec tempor cursus, dui eros pharetra dui, quis tristique dui nulla eget ante.

Curabitur purus enim, blandit at pharetra vestibulum, vestibulum a libero. Integer ligula nibh, dapibus in est sit amet, semper hendrerit purus. Donec mauris neque, dignissim finibus bibendum at, pretium consequat lacus. Ut nec hendrerit libero. Morbi ut dapibus est. Fusce in tempus mi. Morbi dictum sollicitudin metus, eget fermentum massa pellentesque nec. Aliquam eget lorem tempus, lacinia tellus quis, pulvinar lectus. Nullam feugiat lorem eget molestie mattis. Curabitur aliquam arcu sit

CodePen: Offscreen Content → https://codepen.io/aarontgrogg/pen/wBMPMVG

Use cases:

- Hiding/showing navigation menus

Basic implementation:

```
<button popovertarget="menu">
   Toggle Menu
</button>
<nav popover id="menu">
   Nav Content
</nav>
```

```
#menu {
  margin: 0;
  height: 100vh;
  translate: -100vw;
}
#menu:popover-open {
  translate: 0;
}
```

I use a `nav` element to give it semantic value, but you can use any HTML element (`div`, `section`, `aside`, etc.).

A `popover` defaults to `position: fixed` per the User Agent Stylesheet, and is simply pushed off screen when closed, and pulled back onscreen when it is open. Note that `margin: 0` is required if you want to override the User Agent center-alignment.

Clicking outside of the above menu closes it. You can force the panel to stay open, requiring a manual/explicit close, by using `popover="manual"`.

You can also add a `backdrop` pseudo element and style it as you wish:

```
#menu::backdrop {
  background: rgb(190 190 190 / 75%);
}
```

Resources:

- MDN `popover` page → https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/popover
- MDN Popover API examples → https://mdn.github.io/dom-examples/popover-api/
- Introducing the popover API → https://developer.chrome.com/blog/introducing-popover-api/
- Popover API! → https://codepen.io/jh3y/pen/XWPBmmo

Browser compatibility:

- MDN `popover` Browser Compatibility → https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Global_attributes/popover#browser_compatibility

Conclusion

While we all love the power and flexibility JS provides, we should also respect it, and our users, by limiting its use to what it *needs* to do.

There is **so** much more that has changed in recent years, including a ton of options that CSS now covers. If you are now hungry for more, have a look at [my longer article that covers those as well](https://aaront-grogg.com/blog/2023/05/31/replace-js-with-no-js-or-lo-js-options/.

Happy reducing!

Atg

About Aaron T. Grogg

Web Developer / Performance Optimization Specialist

Blog: aarontgrogg.com → https://aarontgrogg.com/

BlueSky: aarontgrogg.bsky.social → https://bsky.app/profile/aarontgrogg.bsky.social

Mastodon: mastodon.social/@aarontgrogg → https://mastodon.social/@aarontgrogg

LinkedIn: linkedin.com/in/aarontgrogg → https://www.linkedin.com/in/aarontgrogg/

Rust Binary Analyis 101 - Part 2

By Mario M.
Published Dec 4, 2025, 12:00 AM
Source: https://mariom3.github.io/malware-analysis/rust-binary-analysis2/

Main Image

Background

Binaries written in Rust have proven notoriously difficult to analyze and in the previous post (Part 1 → https://mariom3.github.io/malware-analysis/rust-binary-analysis1/), we created a basic Rust program to see what all the fuss is about. The program is intended to help us learn how Rust itself compiles commonly encountered malicious code patterns while having access to the source code to facilitate learning. Before diving into analysis of the program we developed, let's look at what makes Rust analysis difficult in general.

Challenges with Rust Analysis

One of the challenges with analyzing Rust binaries is that the binary Rust produces can be very different from the code the developer initially wrote. This means a lot of the logic does not translate in ways we (and our tools) are used to. So when our tooling encounters Rust code, it can result in broken disassembly which further exacerbates the issue. There's a lot of nuanced changes in Rust's ABI that are useful to understand especially when it comes to fixing broken disassembly. I highlight a few points here; however, for further reading, Checkpoint research has a great writeup that dives deep into Rust's features at the binary level: Rust Binary Analysis, Feature by Feature → https://research.checkpoint.com/2023/rust-binary-analysis-feature-by-feature/

Inherent Challenges

At a high level, analysis is complicated by the following inherent challenges which result from the Rust language/compiler and the current state of analysis tools.

- Contiguous strings: This was an initial issue with Rust (and Go) binaries that our tooling did not know how to handle. In my testing, this appears to have been solved in IDA 9.1.
- Aggressive optimization: At times, Rust's aggressive optimization feels like it completely refactors the original developer's code; other times it optimizes it away.
- Functions can access each other's stack frames!
- Broken disassembly: which may result from the previous points discussed along with our current analysis tooling being so C-centric in their design.

Threat Actor Rust Abuse

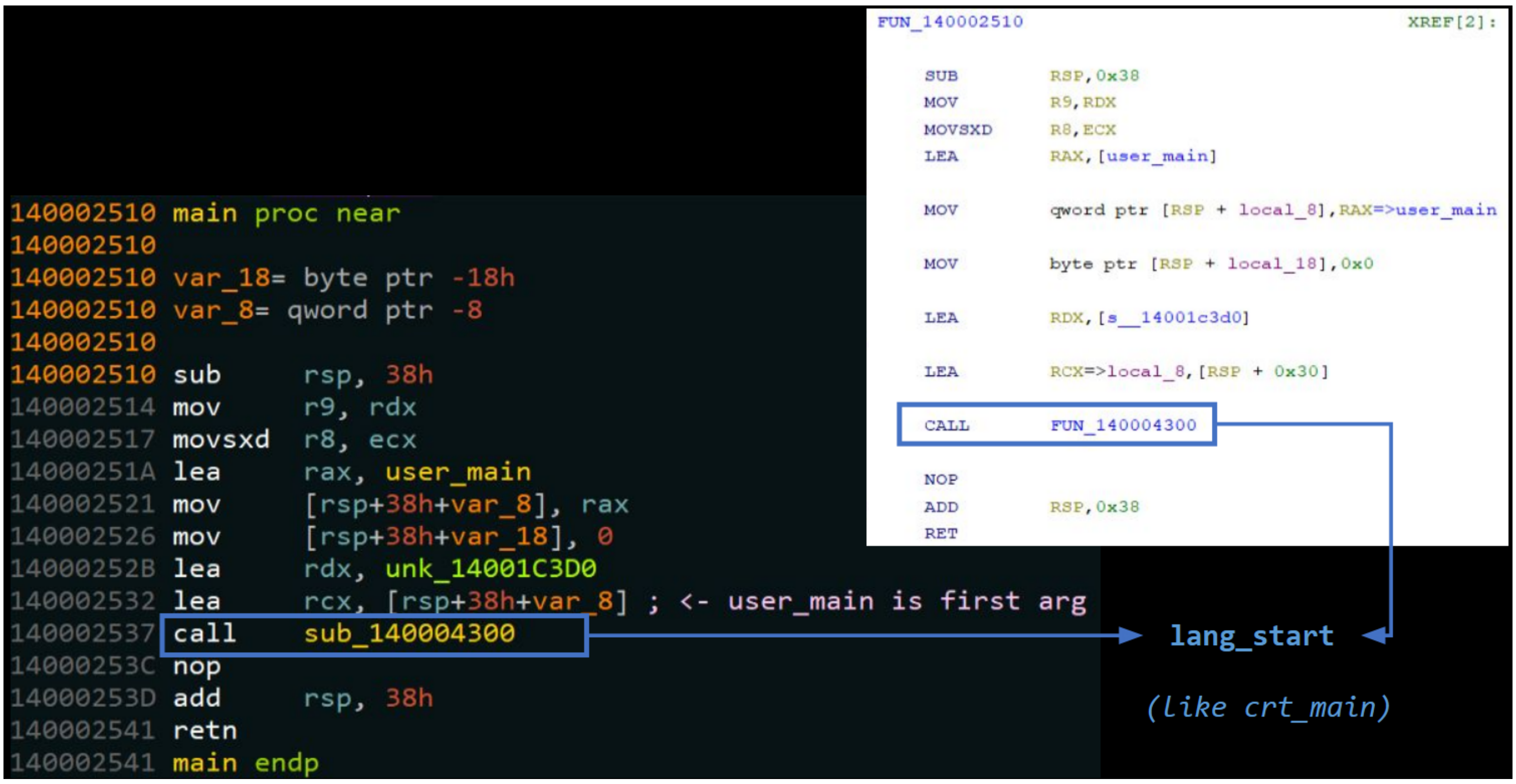
Rust supports many different calling conventions and allows developers to specify them at the function level. Threat actors can abuse this to significantly complicate analysis by creating malware which uses mixed calling conventions.

Rust Binary Analysis (IDA 9.1 vs Ghidra 11.3)

Throughout the rest of this section we'll analyze the program we developed in the previous post (Part 1 → https://mariom3.github.io/malware-analysis/rust-binary-analysis1/) while comparing the output from IDA 9.1 with Ghidra 11.3. If you haven't already, try analyzing the program yourself first as the following will provide a solution.

Finding User Code

When loading up the program we wrote in IDA and Ghidra, neither has an issue disassembling the entry point; which is nothing special. The short function does not directly call the user defined main; instead, it provides it as the 1st argument to the function at `0x140004300`. This function is `lang_start` which is responsible for setting up the Rust runtime before calling our user-defined main.



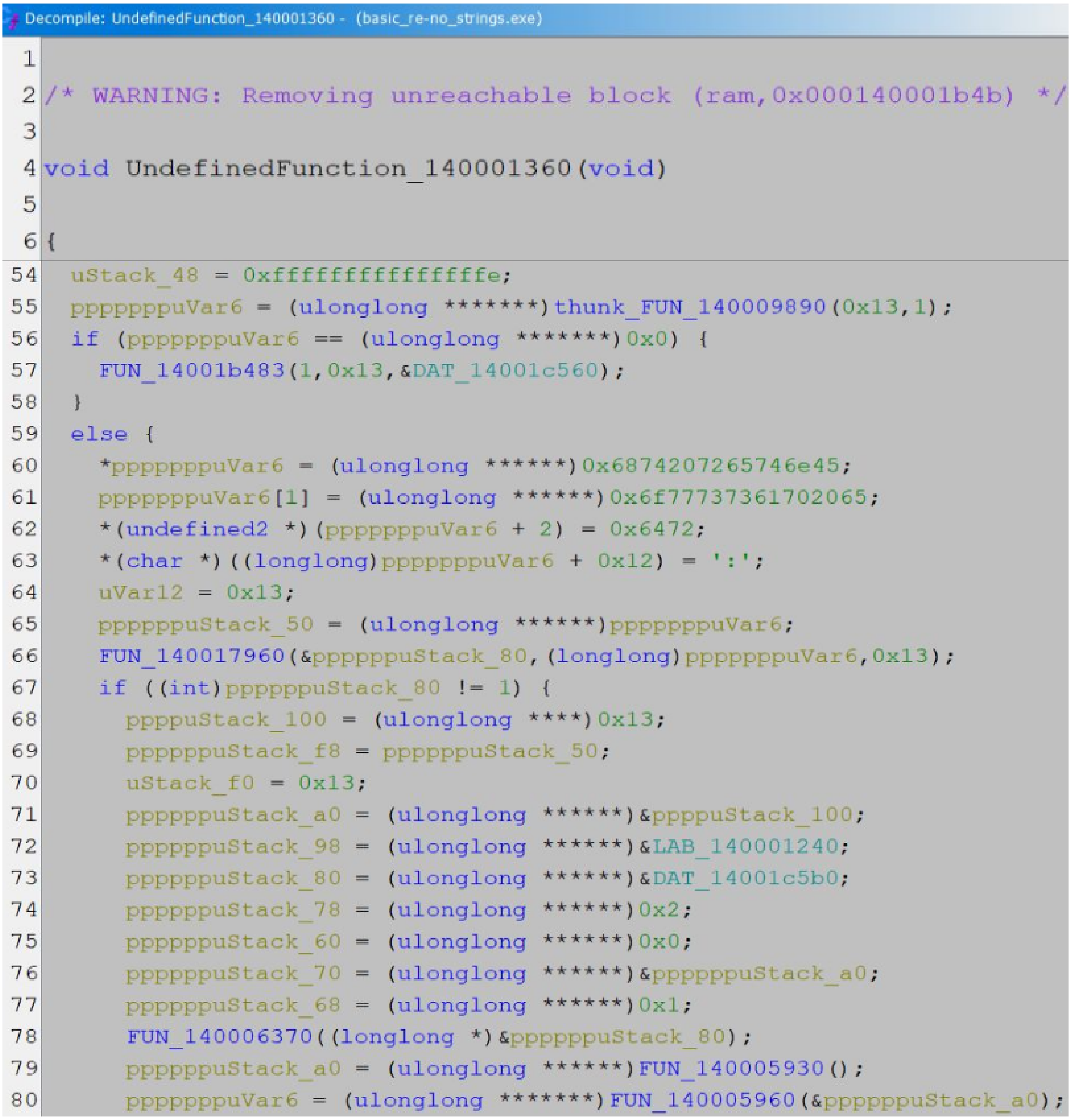
```
FUN_140002510                                            XREF[2]:

     SUB       RSP,0x38
     MOV       R9,RDX
     MOVSXD    R8,ECX
     LEA       RAX,[user_main]
     MOV       qword ptr [RSP + local_8],RAX=>user_main
     MOV       byte ptr [RSP + local_18],0x0
     LEA       RDX,[s__14001c3d0]
     LEA       RCX=>local_8,[RSP + 0x30]
     CALL      FUN_140004300
     NOP
     ADD       RSP,0x38
     RET
```

```
140002510 main proc near
140002510
140002510 var_18= byte ptr -18h
140002510 var_8= qword ptr -8
140002510
140002510 sub     rsp, 38h
140002514 mov     r9, rdx
140002517 movsxd  r8, ecx
14000251A lea     rax, user_main
140002521 mov     [rsp+38h+var_8], rax
140002526 mov     [rsp+38h+var_18], 0
14000252B lea     rdx, unk_14001C3D0
140002532 lea     rcx, [rsp+38h+var_8] ; <- user_main is first arg
140002537 call    sub_140004300
14000253C nop
14000253D add     rsp, 38h
140002541 retn
140002541 main endp
```

Default Entry

Comparing Decompilation of `main`

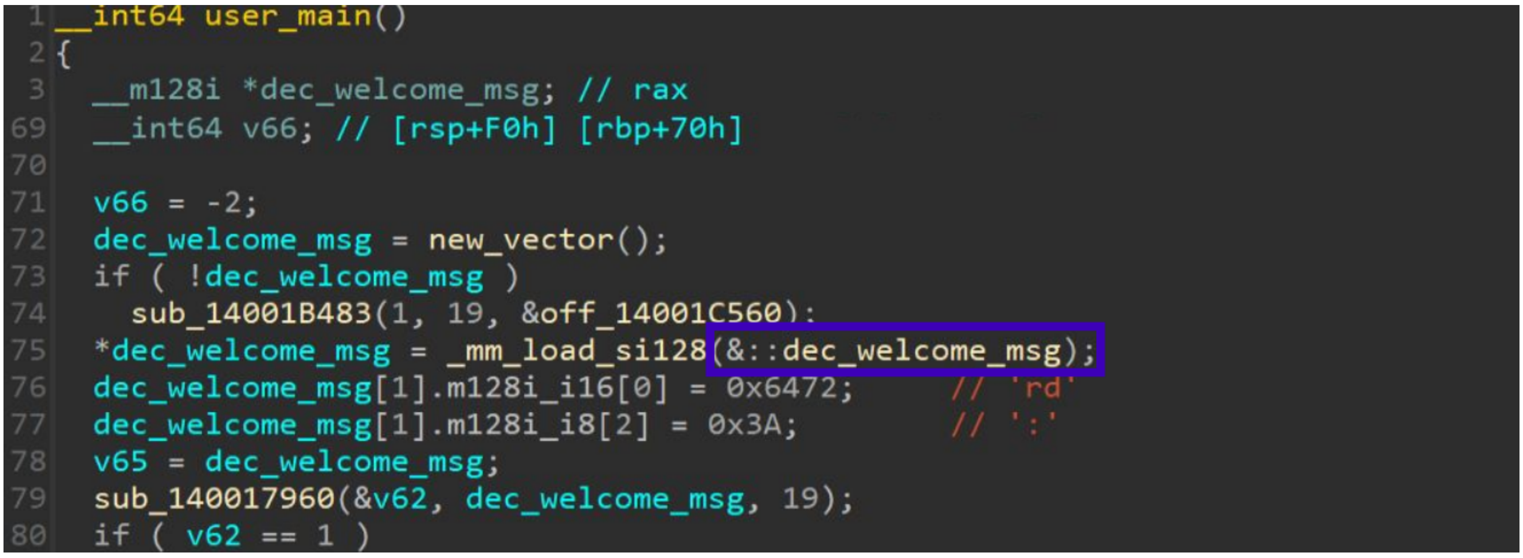
Unfortunately, as soon as we open up Ghidra's decompilation of the user-defined `main`, things are no longer so simple:



```
Decompile: UndefinedFunction_140001360 -  (basic_re-no_strings.exe)
 1
 2 /* WARNING: Removing unreachable block (ram,0x000140001b4b) */
 3
 4 void UndefinedFunction_140001360(void)
 5
 6 {
54   uStack_48 = 0xfffffffffffffffe;
55   ppppppppuVar6 = (ulonglong *******)thunk_FUN_140009890(0x13,1);
56   if (ppppppppuVar6 == (ulonglong *******)0x0) {
57     FUN_14001b483(1,0x13,&DAT_14001c560);
58   }
59   else {
60     *ppppppppuVar6 = (ulonglong ******)0x6874207265746e45;
61     ppppppppuVar6[1] = (ulonglong ******)0x6f77737361702065;
62     *(undefined2 *)(ppppppppuVar6 + 2) = 0x6472;
63     *(char *)((longlong)ppppppppuVar6 + 0x12) = ':';
64     uVar12 = 0x13;
65     pppppppuStack_50 = (ulonglong ******)ppppppppuVar6;
66     FUN_140017960(&pppppppuStack_80,(longlong)ppppppppuVar6,0x13);
67     if ((int)pppppppuStack_80 != 1) {
68       ppppuStack_100 = (ulonglong ****)0x13;
69       pppppppuStack_f8 = pppppppuStack_50;
70       uStack_f0 = 0x13;
71       pppppppuStack_a0 = (ulonglong ******)&ppppuStack_100;
72       pppppppuStack_98 = (ulonglong ******)&LAB_140001240;
73       pppppppuStack_80 = (ulonglong ******)&DAT_14001c5b0;
74       pppppppuStack_78 = (ulonglong ******)0x2;
75       pppppppuStack_60 = (ulonglong ******)0x0;
76       pppppppuStack_70 = (ulonglong ******)&pppppppuStack_a0;
77       pppppppuStack_68 = (ulonglong ******)0x1;
78       FUN_140006370((longlong *)&pppppppuStack_80);
79       pppppppuStack_a0 = (ulonglong ******)FUN_140005930();
80       ppppppppuVar6 = (ulonglong *******)FUN_140005960(&pppppppuStack_a0);
```

Ghidra: Decompilation of main

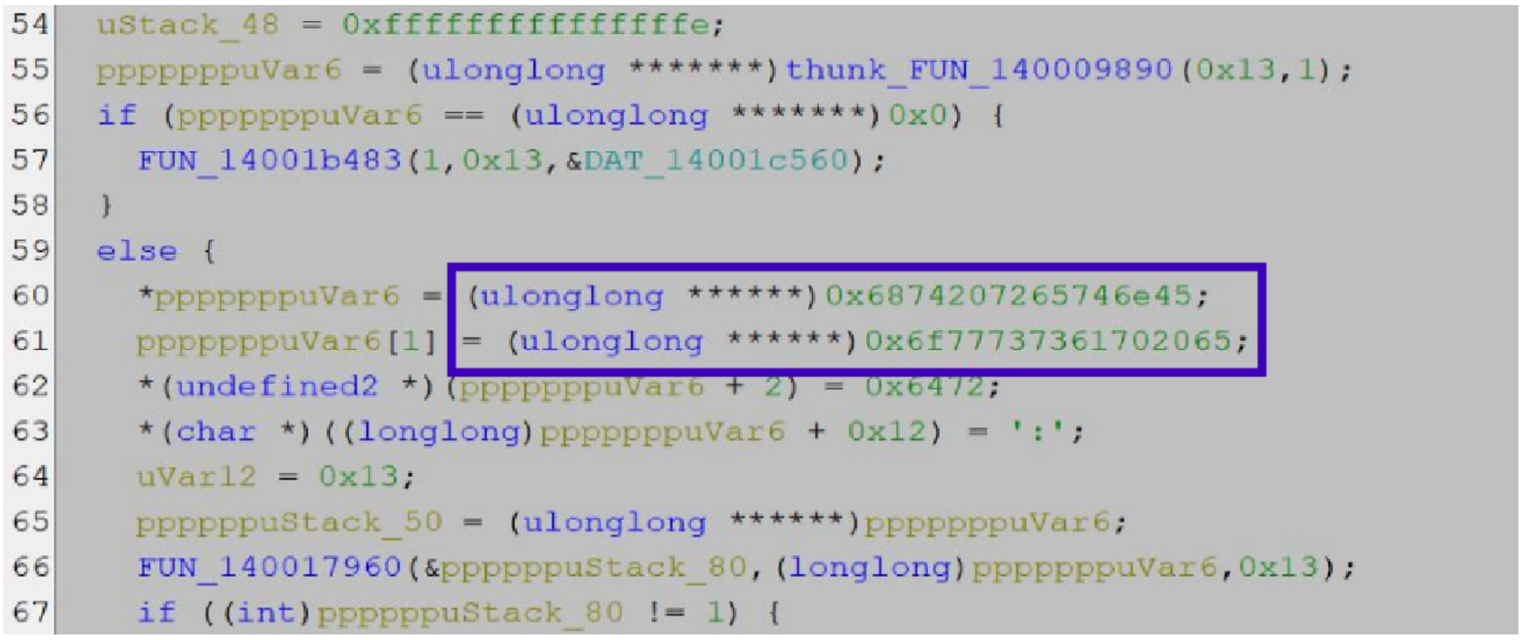
However, IDA handles this much better. Notably, IDA handles the hex arrays we added for our encrypted strings as a simple global variable in line 75:



```
 1 __int64 user_main()
 2 {
 3   __m128i *dec_welcome_msg; // rax
69   __int64 v66; // [rsp+F0h] [rbp+70h]
70
71   v66 = -2;
72   dec_welcome_msg = new_vector();
73   if ( !dec_welcome_msg )
74     sub_14001B483(1, 19, &off_14001C560);
75   *dec_welcome_msg = _mm_load_si128(&::dec_welcome_msg);
76   dec_welcome_msg[1].m128i_i16[0] = 0x6472;      // 'rd'
77   dec_welcome_msg[1].m128i_i8[2] = 0x3A;         // ':'
78   v65 = dec_welcome_msg;
79   sub_140017960(&v62, dec_welcome_msg, 19);
80   if ( v62 == 1 )
```

IDA: Decompilation of main

This is one of the things that makes Ghidra more difficult to understand; it includes the raw hex data in its decompilation. In this case, lines 60-61 correspond to line 75 from IDA:



```
54   uStack_48 = 0xfffffffffffffffe;
55   pppppppuVar6 = (ulonglong *******)thunk_FUN_140009890(0x13,1);
56   if (pppppppuVar6 == (ulonglong *******)0x0) {
57     FUN_14001b483(1,0x13,&DAT_14001c560);
58   }
59   else {
60     *pppppppuVar6 = (ulonglong ******)0x6874207265746e45;
61     pppppppuVar6[1] = (ulonglong ******)0x6f77737361702065;
62     *(undefined2 *)(pppppppuVar6 + 2) = 0x6472;
63     *(char *)((longlong)pppppppuVar6 + 0x12) = ':';
64     uVar12 = 0x13;
65     ppppppuStack_50 = (ulonglong ******)pppppppuVar6;
66     FUN_140017960(&ppppppuStack_80,(longlong)pppppppuVar6,0x13);
67     if ((int)ppppppuStack_80 != 1) {
```

Ghidra: Decompilation of main

As far as our analysis goes, we've found the start of the main portion of the user code. Let's look at it a bit more closely, particularly the hex data from line 75 in IDA's decompilation (60-61 in Ghidra)….

This hex data should be our encrypted welcome message, but if you've stared at hex values long enough… you learn that the majority of ASCII letters correspond with values around `0x60-0x79`. Sooo… is this plaintext? 🤨

We can copy the hex value from the decompilation `6f7773736170206568742072657 46e45` into a tool like CyberChef and we get: `owssap eht retnE`. Right. Endianness. If we reverse it, we get: `"Enter the passwo"`.

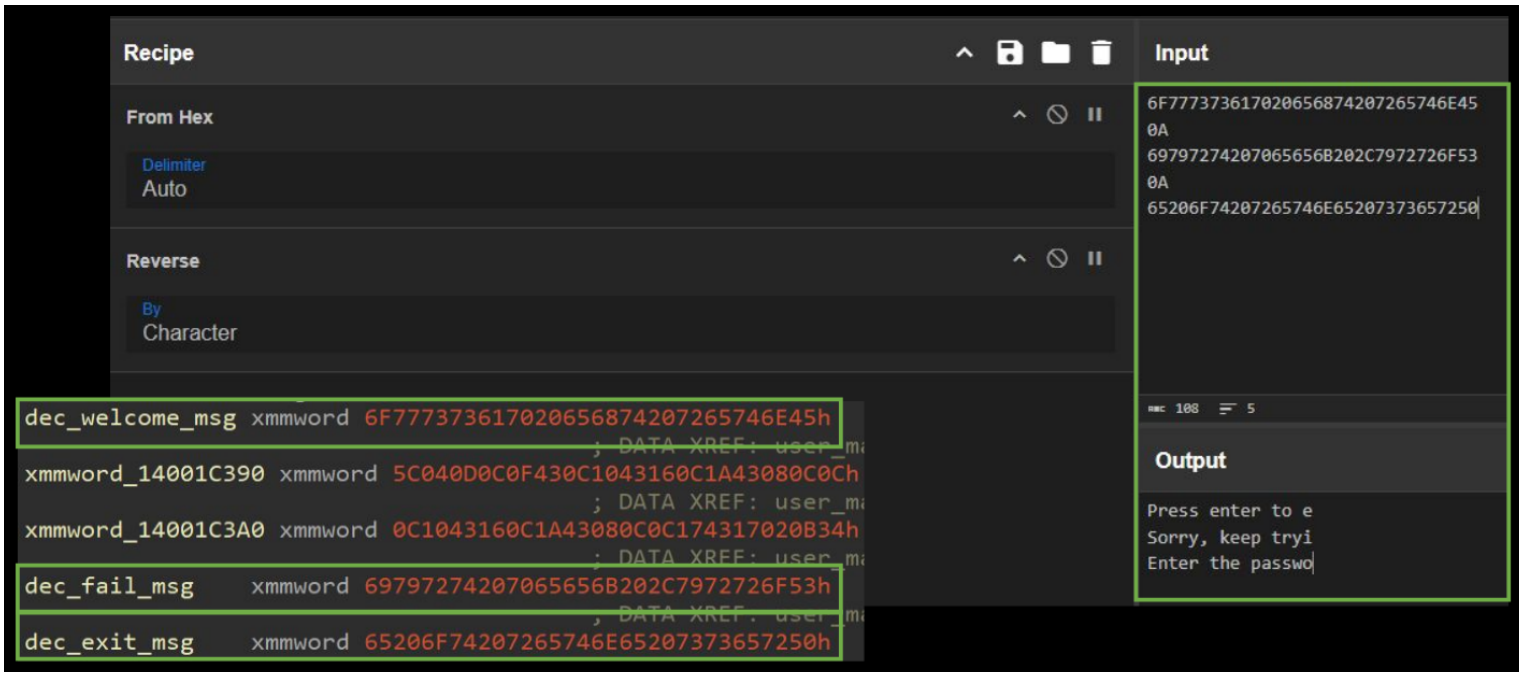
…did I compile the wrong code?

Aggressive Optimizations

I knew the Rust compiler was aggressive with optimizations, but going through this exercise, I learned that it is actually ***really*** aggressive. The Rust compiler decided to execute portions of our code, hard-code the result, and remove the original code we wrote.

Unintended Consequences

Remember when we decided to remove plaintext strings from our code to make analysis a bit more difficult? Well… it turns out, the Rust compiler saw the encrypted strings we hard-coded as byte arrays and decided to decrypt some of them so that it can hard-code the decrypted versions instead! 🫠



```
Delimiter
Auto

By
Character
```

```
6F7773736170206568742072657 46E45
0A
69797274207065656B202C7972726F53
0A
65206F74207265746E65207373657250
```

```
Press enter to e
Sorry, keep tryi
Enter the passwo
```

```
dec_welcome_msg xmmword 6F7773736170206568742072657 46E45h
                                   ; DATA XREF: user_ma
xmmword_14001C390 xmmword 5C040D0C0F430C1043160C1A43080C0Ch
                                   ; DATA XREF: user_ma
xmmword_14001C3A0 xmmword 0C1043160C1A43080C0C174317020B34h
                                   ; DATA XREF: user_ma
dec_fail_msg      xmmword 69797274207065656B202C7972726F53h
                                   ; DATA XREF: user_ma
dec_exit_msg      xmmword 65206F74207265746E65207373657250h
```

Slective Optimization: Decrypted Strings

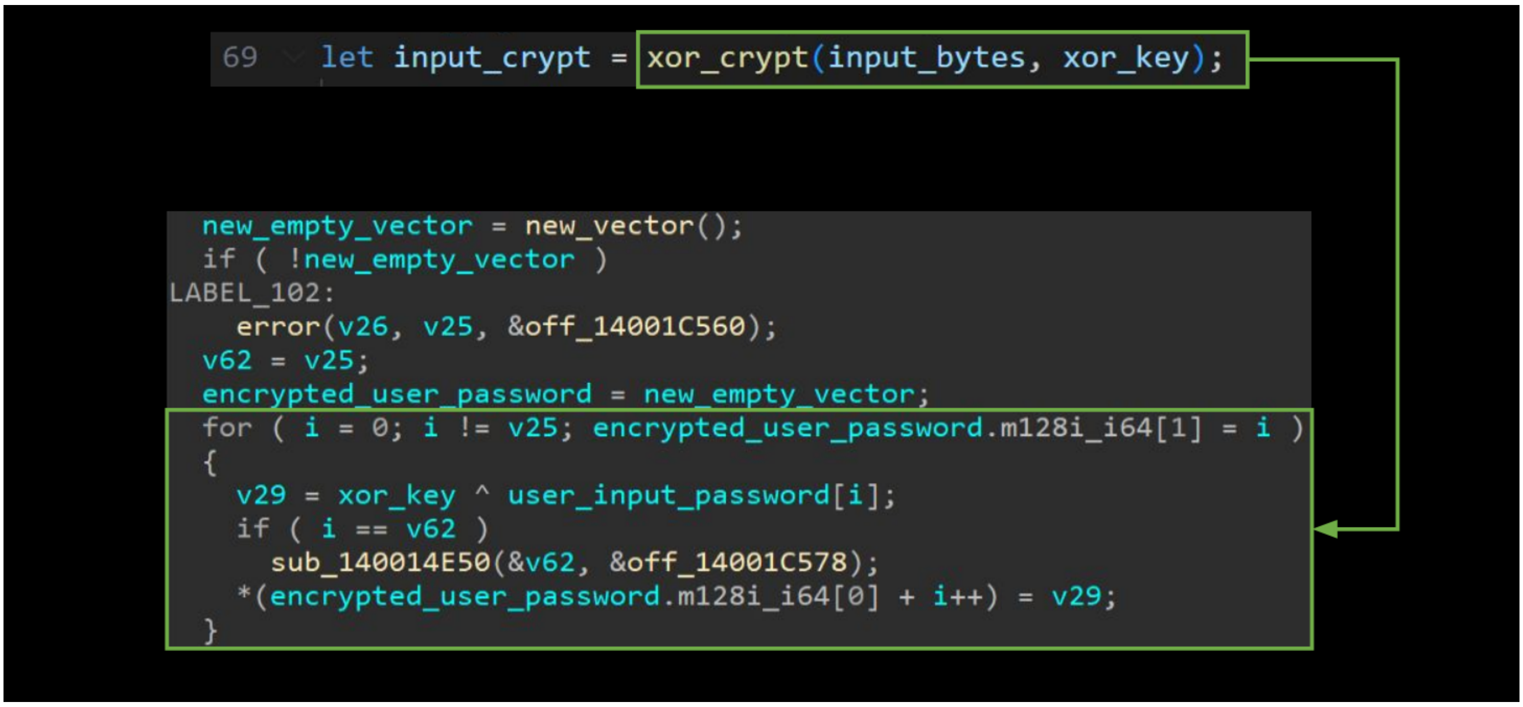
This is pretty surprising, that means that the compiler (***by default***, mind you) not only identified the hard-coded byte arrays, but also found the corresponding `xor_crypt()` call and executed it. I've seen similar behavior other compilers like `gcc`, BUT I had to ***optionally*** increase the aggressiveness of its optimizations.

Rust Knows Better

Another reason your code may look unrecognizable in a disassembler, is that Rust does not preserve the control-flow structure for the programs we write. Remember that helpful `xor_crypt()` function we wrote? Well, Rust's compiler decided to do away with it in our program. Partially because it decrypted some of the strings we were intending to decrypt with it at runtime.

Notably, this did not happen with our password because we were not decrypting it at runtime. Our program still needs the XOR routine to encrypt the user input before comparing it to the hard-coded encrypted password. So what did Rust's compiler do?

It just inlined the logic for `xor_crypt()` instead:



```
let input_crypt = xor_crypt(input_bytes, xor_key);
```

```
  new_empty_vector = new_vector();
  if ( !new_empty_vector )
LABEL_102:
    error(v26, v25, &off_14001C560);
  v62 = v25;
  encrypted_user_password = new_empty_vector;
  for ( i = 0; i != v25; encrypted_user_password.m128i_i64[1] = i )
  {
    v29 = xor_key ^ user_input_password[i];
    if ( i == v62 )
      sub_140014E50(&v62, &off_14001C578);
    *(encrypted_user_password.m128i_i64[0] + i++) = v29;
  }
```

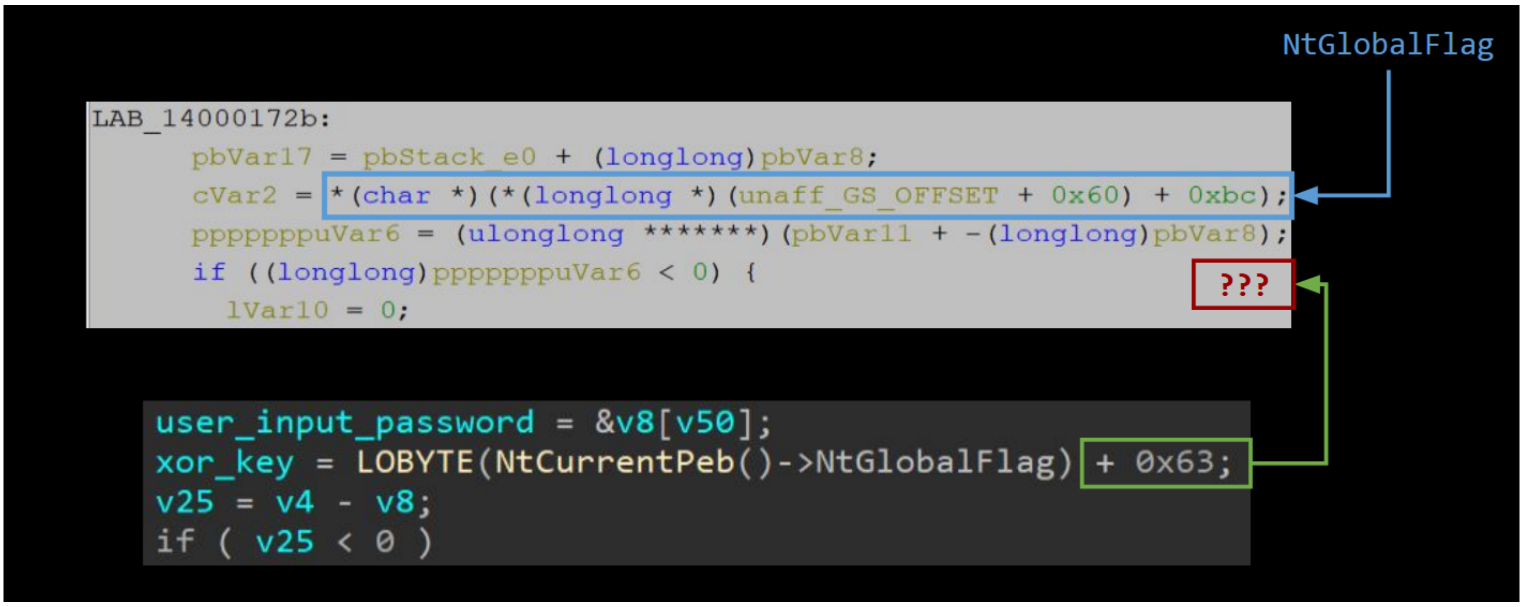
Inlining Functions

Anti-Analysis

Now we know we are at least analyzing the right binary and learned a bit about Rust along the way. If we continue our analysis, we'll eventually come across the anti-analysis functionality we built in.

Anti-Debug Decompilation

Here's how the decompilation for the anti-debug feature we added. Like before, Ghidra's output is not as clean and we have more manual interpretation to do. Such as `unaff_GS_OFFSET + 0x60` being equivalent to `NtCurrent-Peb()`. One oddity with the Ghidra decompilation is that it fails to catch the addition of `0x63` to the `NtGlobalFlag`; it is simply not present:

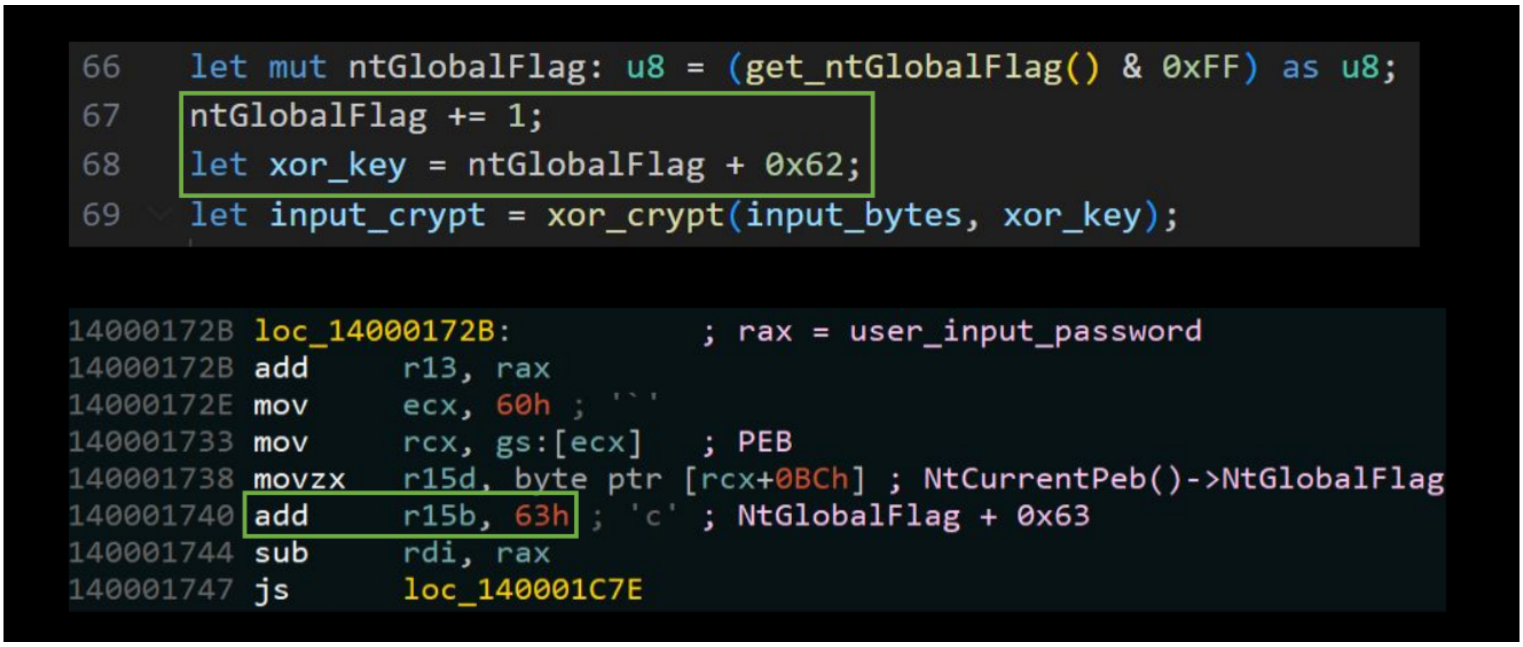


```
LAB_14000172b:
        pbVar17 = pbStack_e0 + (longlong)pbVar8;
        cVar2 = *(char *)(*(longlong *)(unaff_GS_OFFSET + 0x60) + 0xbc);
        pppppppuVar6 = (ulonglong *******)(pbVar11 + -(longlong)pbVar8);
        if ((longlong)pppppppuVar6 < 0) {
          lVar10 = 0;
```

```
user_input_password = &v8[v50];
xor_key = LOBYTE(NtCurrentPeb()->NtGlobalFlag) + 0x63;
v25 = v4 - v8;
if ( v25 < 0 )
```

IDA vs Ghidra Decompilation of Anti-Debug

This decompilation corresponds to the code where we dynamically set the XOR key using the `NtGlobalFlag` variable. We also see another, but not surprising, optimization: `ntGlobalFlag +=1; xor_key = ntGlobalFlag + 0x62;` simply becomes `xor_key = ntGlobalFlag + 0x63;`. Having analyzed this, we'll know the XOR key for the password is **`0x63`**! This is shown below using the IDA disassembly, but it's the same on both tools:



```
let mut ntGlobalFlag: u8 = (get_ntGlobalFlag() & 0xFF) as u8;
ntGlobalFlag += 1;
let xor_key = ntGlobalFlag + 0x62;
let input_crypt = xor_crypt(input_bytes, xor_key);
```

```
14000172B loc_14000172B:                ; rax = user_input_password
14000172B add      r13, rax
14000172E mov      ecx, 60h ; '`'
140001733 mov      rcx, gs:[ecx]     ; PEB
140001738 movzx    r15d, byte ptr [rcx+0BCh] ; NtCurrentPeb()->NtGlobalFlag
140001740 add      r15b, 63h ; 'c'   ; NtGlobalFlag + 0x63
140001744 sub      rdi, rax
140001747 js       loc_140001C7E
```

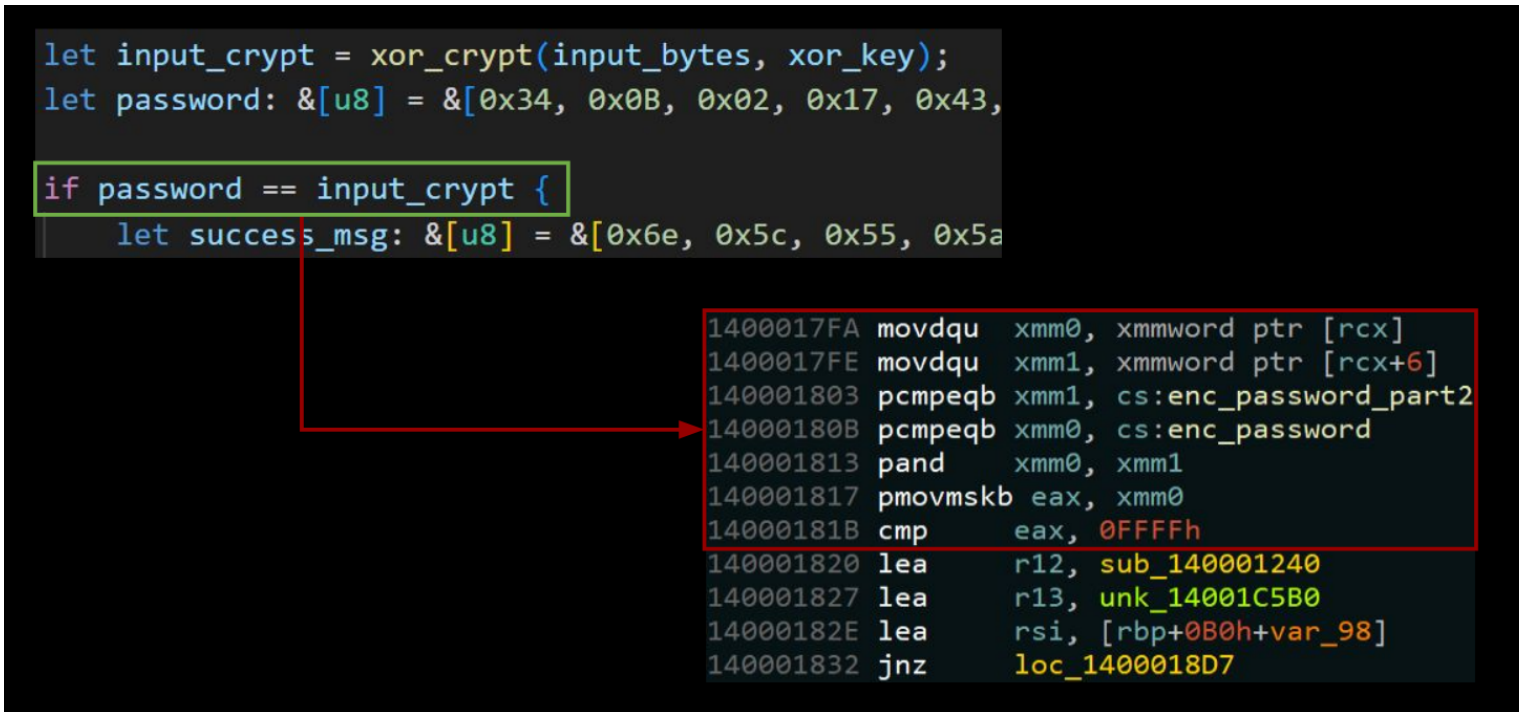
Corresponding Source Code

At this point we're pretty close to finding the password! Let's continue following this code's execution flow.

Rust LOVES SSE Instructions

At this point we find the logic for comparing the encrypted password with the XOR'ed user input! Unfortunately, this logic manifests another *quirk* of Rust… 🙂 its love for SSE instructions! Personally, I think it's just optimizing for maximum headaches in reverse engineers… but, what do I know?

Here's the disassembly with the corresponding source code we wrote:



```
let input_crypt = xor_crypt(input_bytes, xor_key);
let password: &[u8] = &[0x34, 0x0B, 0x02, 0x17, 0x43,

if password == input_crypt {
    let success_msg: &[u8] = &[0x6e, 0x5c, 0x55, 0x5a
```

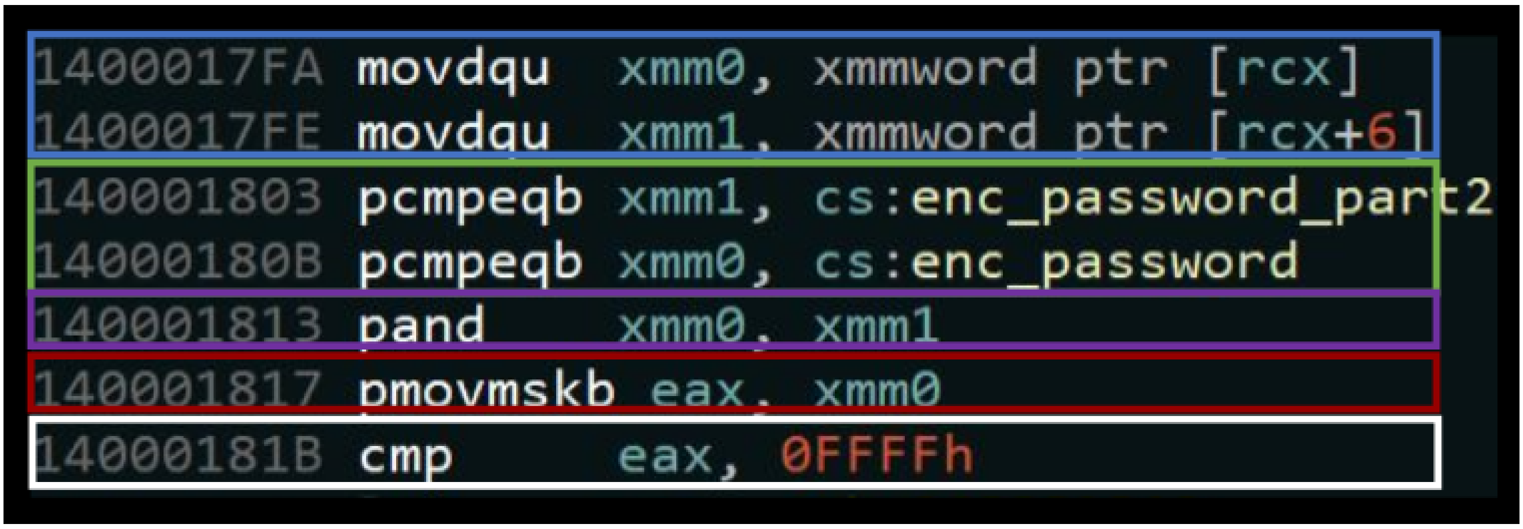
```
1400017FA movdqu   xmm0, xmmword ptr [rcx]
1400017FE movdqu   xmm1, xmmword ptr [rcx+6]
140001803 pcmpeqb  xmm1, cs:enc_password_part2
14000180B pcmpeqb  xmm0, cs:enc_password
140001813 pand     xmm0, xmm1
140001817 pmovmskb eax, xmm0
14000181B cmp      eax, 0FFFFh
140001820 lea      r12, sub_140001240
140001827 lea      r13, unk_14001C5B0
14000182E lea      rsi, [rbp+0B0h+var_98]
140001832 jnz      loc_1400018D7
```

Covercomplicating it with SSE

Let's break down these SSE instructions. Honestly, it's not that bad, it's a matter of gaining familiarity with new/uncommon instructions. Below is a color coded breakdown of what the extensions do:

1. Load the user password parts in `xmm0` and `xmm1`.
2. Compare with the expected values from `.rdata` (encrypted password).
3. 128-bit AND the results from each comparison.

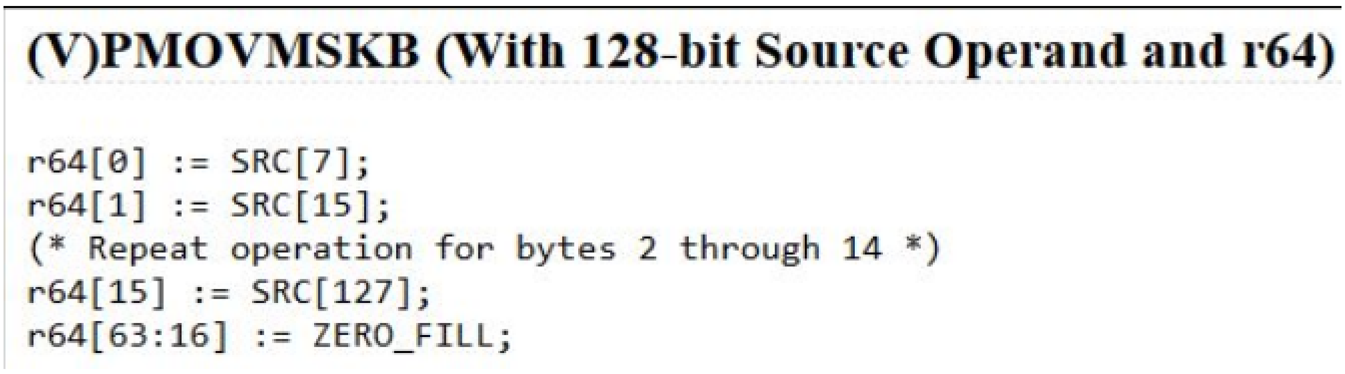
4. ???
5. Compare the result. (Success if `eax != 0x0FFF`).



```
1400017FA movdqu   xmm0, xmmword ptr [rcx]
1400017FE movdqu   xmm1, xmmword ptr [rcx+6]
140001803 pcmpeqb xmm1, cs:enc_password_part2
14000180B pcmpeqb xmm0, cs:enc_password
140001813 pand     xmm0, xmm1
140001817 pmovmskb eax, xmm0
14000181B cmp      eax, 0FFFFh
```

Color Coded SSE Instructions

Ok, the `pmovmskb` needed a bit more room for an explanation. At least it was a bit more difficult for me to wrap my head around initially. Especially when trying to work with a definition like this.



(V)PMOVMSKB (With 128-bit Source Operand and r64)

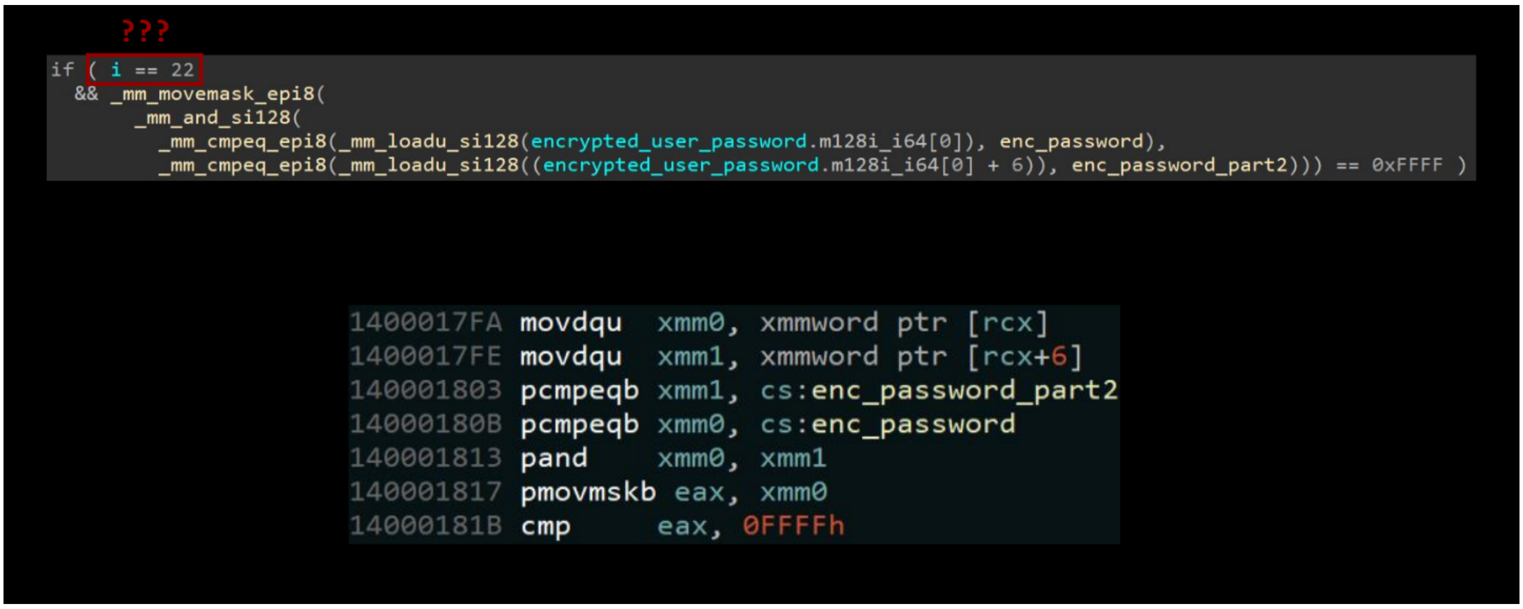
```
r64[0] := SRC[7];
r64[1] := SRC[15];
(* Repeat operation for bytes 2 through 14 *)
r64[15] := SRC[127];
r64[63:16] := ZERO_FILL;
```

Screenshot of pmovmskb Definition

After consulting with my therapist — I mean — ChatGPT, I came to understand that the instruction `pmovmskb` extracts the most significant bit from every 8 bits of `xmm0`. Since an `xmm` register is 128-bits, the result is 16 bits (4-bytes), hence the final comparison to the 4-byte value `0xFFFF`.

Cool, but how does the decompilation look like?

In this case, both struggled. Below we'll see IDA's decompilation which seemingly randomly includes a comparison `i == 22`? Otherwise, it's not too difficult to read.

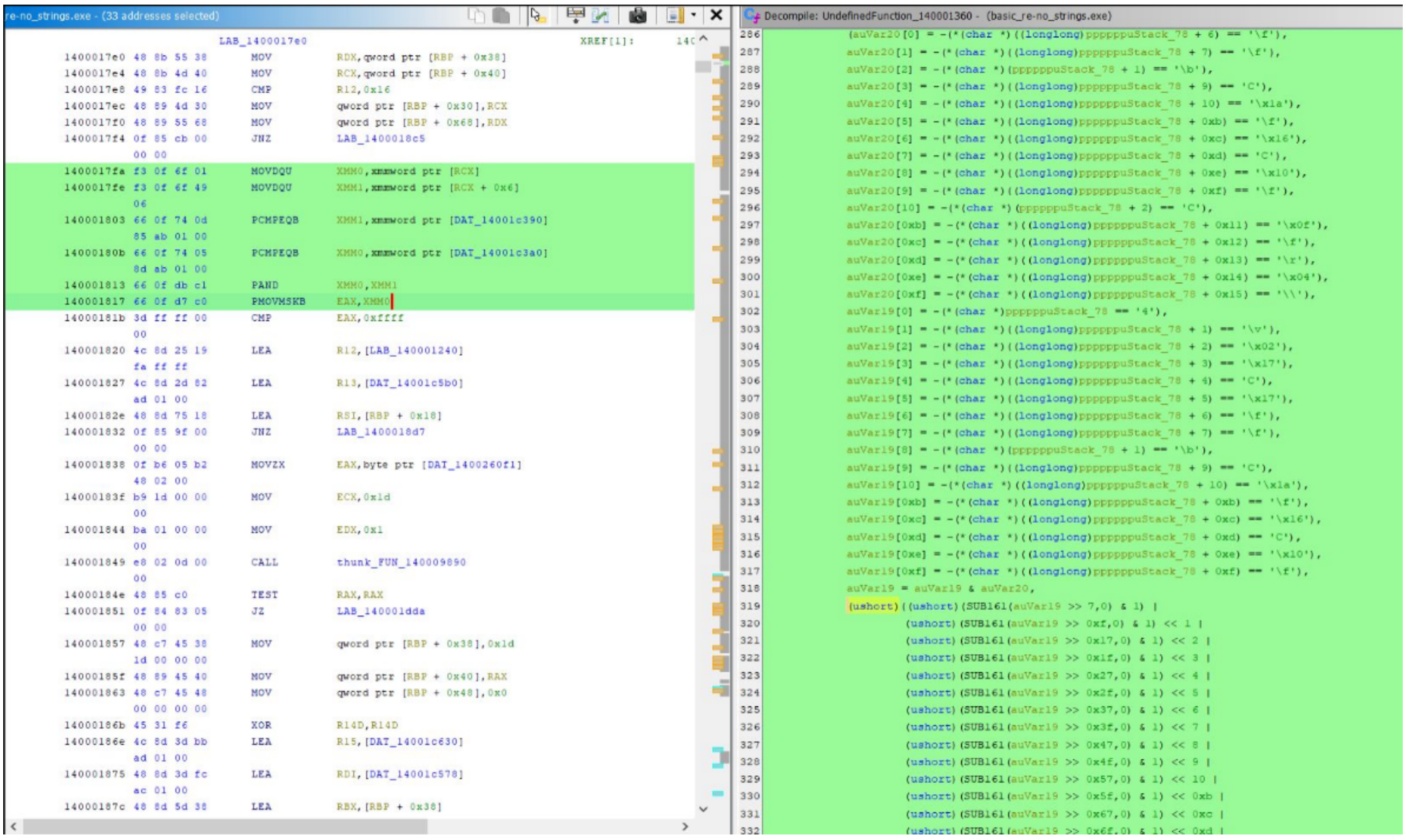


```
if ( i == 22
  && _mm_movemask_epi8(
       _mm_and_si128(
         _mm_cmpeq_epi8(_mm_loadu_si128(encrypted_user_password.m128i_i64[0]), enc_password),
         _mm_cmpeq_epi8(_mm_loadu_si128((encrypted_user_password.m128i_i64[0] + 6)), enc_password_part2))) == 0xFFFF )
```

```
1400017FA movdqu   xmm0, xmmword ptr [rcx]
1400017FE movdqu   xmm1, xmmword ptr [rcx+6]
140001803 pcmpeqb  xmm1, cs:enc_password_part2
14000180B pcmpeqb  xmm0, cs:enc_password
140001813 pand     xmm0, xmm1
140001817 pmovmskb eax, xmm0
14000181B cmp      eax, 0FFFFh
```

IDA Decompilation

Ghidra on the other hand… had a bit more of a hard time figuring out how to decompile SSE instructions. 🥴😵


re-no_strings.exe - (33 addresses selected)
Decompile: UndefinedFunction_140001360 - (basic_re-no_strings.exe)

```
                             LAB_1400017e0                                   XREF[1]:     14(
        1400017e0 48 8b 55 38       MOV        RDX,qword ptr [RBP + 0x38]
        1400017e4 48 8b 4d 40       MOV        RCX,qword ptr [RBP + 0x40]
        1400017e8 49 83 fc 16       CMP        R12,0x16
        1400017ec 48 89 4d 30       MOV        qword ptr [RBP + 0x30],RCX
        1400017f0 48 89 55 68       MOV        qword ptr [RBP + 0x68],RDX
        1400017f4 0f 85 cb 00       JNZ        LAB_1400018c5
                  00 00
        1400017fa f3 0f 6f 01       MOVDQU     XMM0,xmmword ptr [RCX]
        1400017fe f3 0f 6f 49       MOVDQU     XMM1,xmmword ptr [RCX + 0x6]
                  06
        140001803 66 0f 74 0d       PCMPEQB    XMM1,xmmword ptr [DAT_14001c390]
                  85 ab 01 00
        14000180b 66 0f 74 05       PCMPEQB    XMM0,xmmword ptr [DAT_14001c3a0]
                  8d ab 01 00
        140001813 66 0f db c1       PAND       XMM0,XMM1
        140001817 66 0f d7 c0       PMOVMSKB   EAX,XMM0
        14000181b 3d ff ff 00       CMP        EAX,0xffff
                  00
        140001820 4c 8d 25 19       LEA        R12,[LAB_140001240]
                  fa ff ff
        140001827 4c 8d 2d 82       LEA        R13,[DAT_14001c5b0]
                  ad 01 00
        14000182e 48 8d 75 18       LEA        RSI,[RBP + 0x18]
        140001832 0f 85 9f 00       JNZ        LAB_1400018d7
                  00 00
        140001838 0f b6 05 b2       MOVZX      EAX,byte ptr [DAT_1400260f1]
                  48 02 00
        14000183f b9 1d 00 00       MOV        ECX,0x1d
                  00
        140001844 ba 01 00 00       MOV        EDX,0x1
                  00
        140001849 e8 02 0d 00       CALL       thunk_FUN_140009890
                  00
        14000184e 48 85 c0          TEST       RAX,RAX
        140001851 0f 84 83 05       JZ         LAB_140001dda
                  00 00
        140001857 48 c7 45 38       MOV        qword ptr [RBP + 0x38],0x1d
                  1d 00 00 00
        14000185f 48 89 45 40       MOV        qword ptr [RBP + 0x40],RAX
        140001863 48 c7 45 48       MOV        qword ptr [RBP + 0x48],0x0
                  00 00 00 00
        14000186b 45 31 f6          XOR        R14D,R14D
        14000186e 4c 8d 3d bb       LEA        R15,[DAT_14001c630]
                  ad 01 00
        140001875 48 8d 3d fc       LEA        RDI,[DAT_14001c578]
                  ac 01 00
        14000187c 48 8d 5d 38       LEA        RBX,[RBP + 0x38]
```

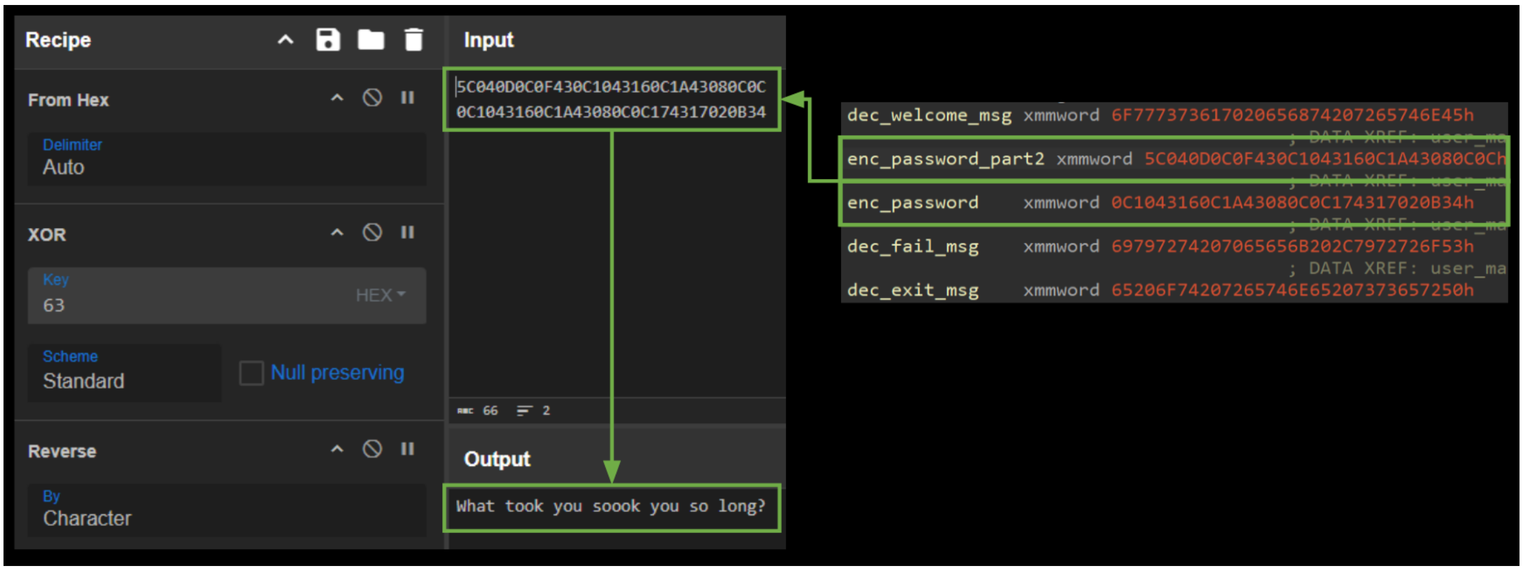
```
          (auVar20[0] = -(*(char *)((longlong)ppppppuStack_78 + 6) == '\f'),
          auVar20[1] = -(*(char *)((longlong)ppppppuStack_78 + 7) == '\f'),
          auVar20[2] = -(*(char *)(ppppppuStack_78 + 1) == '\b'),
          auVar20[3] = -(*(char *)((longlong)ppppppuStack_78 + 9) == 'C'),
          auVar20[4] = -(*(char *)((longlong)ppppppuStack_78 + 10) == '\x1a'),
          auVar20[5] = -(*(char *)((longlong)ppppppuStack_78 + 0xb) == '\f'),
          auVar20[6] = -(*(char *)((longlong)ppppppuStack_78 + 0xc) == '\x16'),
          auVar20[7] = -(*(char *)((longlong)ppppppuStack_78 + 0xd) == 'C'),
          auVar20[8] = -(*(char *)((longlong)ppppppuStack_78 + 0xe) == '\x10'),
          auVar20[9] = -(*(char *)((longlong)ppppppuStack_78 + 0xf) == '\f'),
          auVar20[10] = -(*(char *)(ppppppuStack_78 + 2) == 'C'),
          auVar20[0xb] = -(*(char *)((longlong)ppppppuStack_78 + 0x11) == '\x0f'),
          auVar20[0xc] = -(*(char *)((longlong)ppppppuStack_78 + 0x12) == '\f'),
          auVar20[0xd] = -(*(char *)((longlong)ppppppuStack_78 + 0x13) == '\r'),
          auVar20[0xe] = -(*(char *)((longlong)ppppppuStack_78 + 0x14) == '\x04'),
          auVar20[0xf] = -(*(char *)((longlong)ppppppuStack_78 + 0x15) == '\\'),
          auVar19[0] = -(*(char *)ppppppuStack_78 == '4'),
          auVar19[1] = -(*(char *)((longlong)ppppppuStack_78 + 1) == '\v'),
          auVar19[2] = -(*(char *)((longlong)ppppppuStack_78 + 2) == '\x02'),
          auVar19[3] = -(*(char *)((longlong)ppppppuStack_78 + 3) == '\x17'),
          auVar19[4] = -(*(char *)((longlong)ppppppuStack_78 + 4) == 'C'),
          auVar19[5] = -(*(char *)((longlong)ppppppuStack_78 + 5) == '\x17'),
          auVar19[6] = -(*(char *)((longlong)ppppppuStack_78 + 6) == '\f'),
          auVar19[7] = -(*(char *)((longlong)ppppppuStack_78 + 7) == '\f'),
          auVar19[8] = -(*(char *)(ppppppuStack_78 + 1) == '\b'),
          auVar19[9] = -(*(char *)((longlong)ppppppuStack_78 + 9) == 'C'),
          auVar19[10] = -(*(char *)((longlong)ppppppuStack_78 + 10) == '\x1a'),
          auVar19[0xb] = -(*(char *)((longlong)ppppppuStack_78 + 0xb) == '\f'),
          auVar19[0xc] = -(*(char *)((longlong)ppppppuStack_78 + 0xc) == '\x16'),
          auVar19[0xd] = -(*(char *)((longlong)ppppppuStack_78 + 0xd) == 'C'),
          auVar19[0xe] = -(*(char *)((longlong)ppppppuStack_78 + 0xe) == '\x10'),
          auVar19[0xf] = -(*(char *)((longlong)ppppppuStack_78 + 0xf) == '\f'),
          auVar19 = auVar19 & auVar20,
          (ushort)((ushort)(SUB161(auVar19 >> 7,0) & 1) |
                  (ushort)(SUB161(auVar19 >> 0xf,0) & 1) << 1 |
                  (ushort)(SUB161(auVar19 >> 0x17,0) & 1) << 2 |
                  (ushort)(SUB161(auVar19 >> 0x1f,0) & 1) << 3 |
                  (ushort)(SUB161(auVar19 >> 0x27,0) & 1) << 4 |
                  (ushort)(SUB161(auVar19 >> 0x2f,0) & 1) << 5 |
                  (ushort)(SUB161(auVar19 >> 0x37,0) & 1) << 6 |
                  (ushort)(SUB161(auVar19 >> 0x3f,0) & 1) << 7 |
                  (ushort)(SUB161(auVar19 >> 0x47,0) & 1) << 8 |
                  (ushort)(SUB161(auVar19 >> 0x4f,0) & 1) << 9 |
                  (ushort)(SUB161(auVar19 >> 0x57,0) & 1) << 10 |
                  (ushort)(SUB161(auVar19 >> 0x5f,0) & 1) << 0xb |
                  (ushort)(SUB161(auVar19 >> 0x67,0) & 1) << 0xc |
                  (ushort)(SUB161(auVar19 >> 0x6f,0) & 1) << 0xd |
```

Ghidra Decompilation

Password Recovery

Now that we understand what the SSE instructions are doing, we know that the hex data from `.rdata` is the encrypted password… and we have the XOR key from our previous analysis of the anti-debug functionality!

Now we can simply copy the hex data used to compare to the XOR'ed user input into a tool like CyberChef and… we've got the password! 🎉



Recipe

From Hex
Delimiter
Auto

XOR
Key
63 HEX
Scheme
Standard
Null preserving

Reverse
By
Character

Input

```
5C040D0C0F430C1043160C1A43080C0C
0C1043160C1A43080C0C174317020B34
```

Output

```
What took you soook you so long?
```

```
dec_welcome_msg xmmword 6F777373617020656874207265746E45h
                                              ; DATA XREF: user_ma
enc_password_part2 xmmword 5C040D0C0F430C1043160C1A43080C0Ch
                                              ; DATA XREF: user_ma
enc_password     xmmword 0C1043160C1A43080C0C174317020B34h
                                              ; DATA XREF: user_ma
dec_fail_msg     xmmword 69797274207065656B202C7972726F53h
                                              ; DATA XREF: user_ma
dec_exit_msg     xmmword 65206F74207265746E65207373657250h
```

Recovered Password

Helpful Resources

- CheckPoint Research Deep-Dive: Rust Binary Analysis, Feature by Feature → https://research.checkpoint.com/2023/rust-binary-analysis-feature-by-feature/
- BlackHat Talk on the Rust Malware Ecosystem: Rust Malware Ecosystem → https://www.youtube.com/watch?v=cMlhIARmNfU

mprocs: start all your project's commands at once

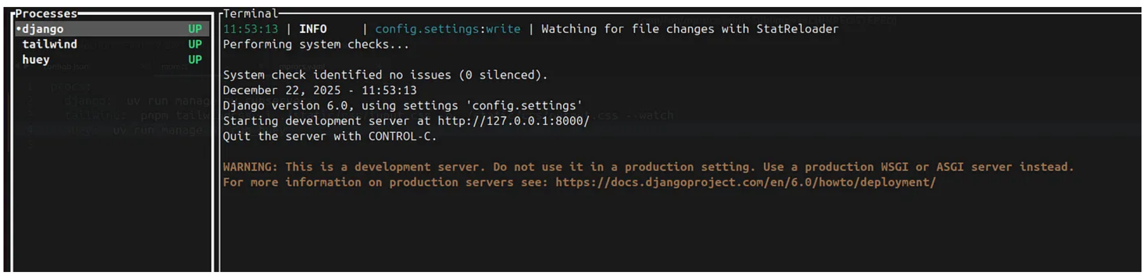
By Bite Code!
Published Dec 22, 2025, 11:06 PM
Source: https://www.bitecode.dev/p/mprocs-start-all-your-projects-commands

mprocs → https://github.com/pvolok/mprocs is a small dev-oriented process manager that lets you run and monitor long-running processes in the background.

You install → https://github.com/pvolok/mprocs?tab=readme-ov-file#installation it (or just put the binary on your path), then create a `mprocs.yaml` file at the root of your project with your commands:

```
procs:
  django: uv run manage.py runserver
  tailwind: npx @tailwindcss/cli -i input.css -o output.css -w
  huey: uv run huey_consumer.py my_app.huey -k process -w 4
```

Run `mprocs` so that it starts all of them at once, then shows you a nice TUI to follow up.



```
Processes
•django        UP
 tailwind      UP
 huey          UP

Terminal
11:53:13 | INFO     | config.settings:write | Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
December 22, 2025 - 11:53:13
Django version 6.0, using settings 'config.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.

WARNING: This is a development server. Do not use it in a production setting. Use a production WSGI or ASGI server instead.
For more information on production servers see: https://docs.djangoproject.com/en/6.0/howto/deployment/
```

I just encountered a post from Hynek → https://bsky.app/profile/hynek.me/post/3magfqaymu22a mentioning mprocs → https://github.com/pvolok/mprocs, and it looked like something I wanted for a long time.

Indeed, for many of my projects, I have multiple long-running processes I need to run, like a web server, a task queue, a CSS pre-processor, and so on.

Starting and monitoring each of them in a different terminal tab every time, and restarting them when they fail, is slightly annoying. I say slightly, because we are clearly in first-world problem territory. But annoying nonetheless.

However, not annoying enough that I wanted to deal with the hassle of using `tmux` just for this.

Turns out `mprocs` is exactly what I want, and maybe that's what you want too.

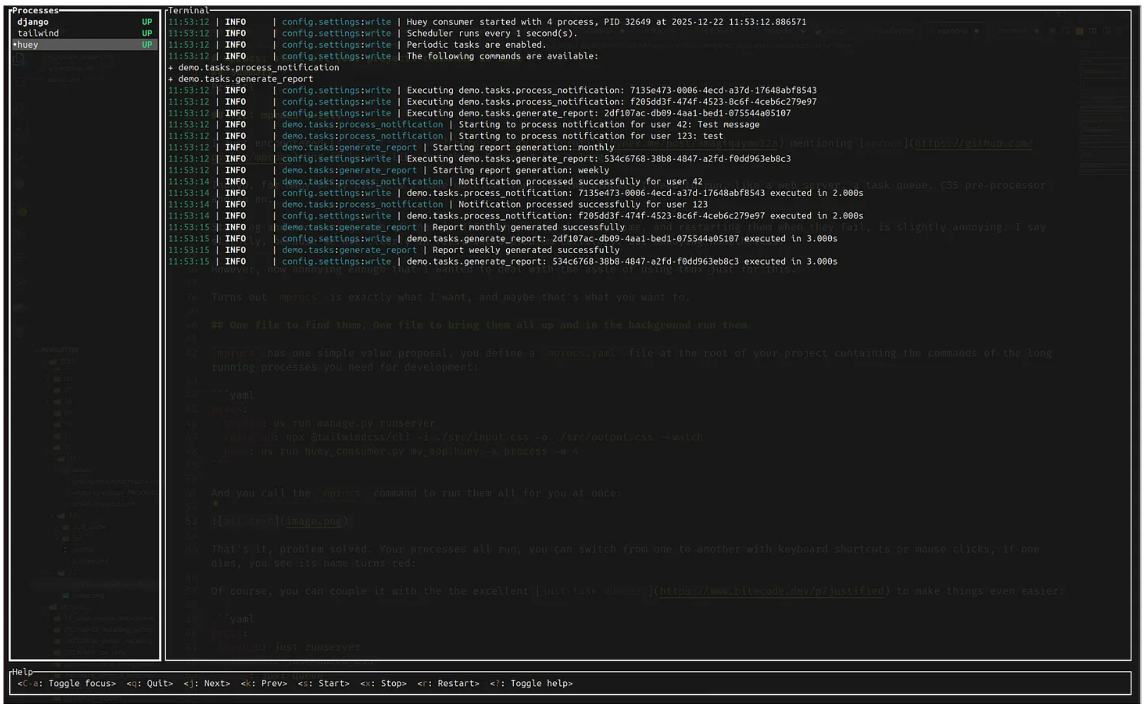
It has one simple value proposal: you define a `mprocs.yaml` file at the root of your project containing the commands of the long-running processes you need for development:

```
procs:
  command1: run your server
  command2: run your builder
  commadn3: run whatever you want really
```

E.G, for a Python Web project with a bit of Tailwind:

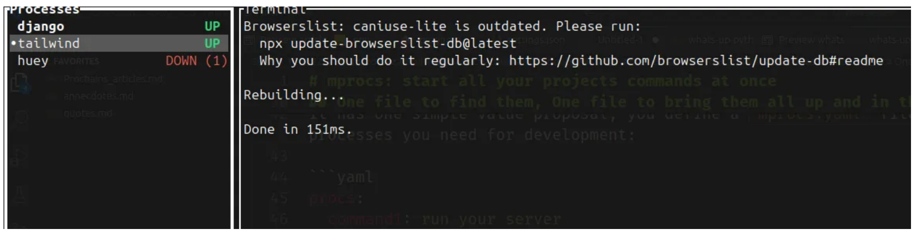
```
procs:
  django: uv run manage.py runserver
  tailwind: npx @tailwindcss/cli -i .input.css -o .output.css -w
  huey: uv run huey_consumer.py my_app.huey -k process -w 4
```

And you call the `mprocs` command to run them all for you at once:


Processes
django UP
tailwind UP
•huey UP
Terminal
Help
<C-a: Toggle focus> <q: Quit> <j: Next> <k: Prev> <s: Start> <x: Stop> <r: Restart> <?: Toggle help>

```
11:53:12 | INFO     | config.settings:write | Huey consumer started with 4 process, PID 32649 at 2025-12-22 11:53:12.886571
11:53:12 | INFO     | config.settings:write | Scheduler runs every 1 second(s).
11:53:12 | INFO     | config.settings:write | Periodic tasks are enabled.
11:53:12 | INFO     | config.settings:write | The following commands are available:
+ demo.tasks.process_notification
+ demo.tasks.generate_report
11:53:12 | INFO     | config.settings:write | Executing demo.tasks.process_notification: 7135e473-0006-4ecd-a37d-17648abf8543
11:53:12 | INFO     | config.settings:write | Executing demo.tasks.process_notification: f205dd3f-474f-4523-8c6f-4ceb6c279e97
11:53:12 | INFO     | config.settings:write | Executing demo.tasks.generate_report: 2df107ac-db09-4aa1-bed1-075544a05107
11:53:12 | INFO     | demo.tasks:process_notification | Starting to process notification for user 42: Test message
11:53:12 | INFO     | demo.tasks:process_notification | Starting to process notification for user 123: test
11:53:12 | INFO     | demo.tasks:generate_report | Starting report generation: monthly
11:53:12 | INFO     | config.settings:write | Executing demo.tasks.generate_report: 534c6768-38b8-4847-a2fd-f0dd963eb8c3
11:53:12 | INFO     | demo.tasks:generate_report | Starting report generation: weekly
11:53:14 | INFO     | demo.tasks:process_notification | Notification processed successfully for user 42
11:53:14 | INFO     | config.settings:write | demo.tasks.process_notification: 7135e473-0006-4ecd-a37d-17648abf8543 executed in 2.000s
11:53:14 | INFO     | demo.tasks:process_notification | Notification processed successfully for user 123
11:53:14 | INFO     | config.settings:write | demo.tasks.process_notification: f205dd3f-474f-4523-8c6f-4ceb6c279e97 executed in 2.000s
11:53:15 | INFO     | demo.tasks:generate_report | Report monthly generated successfully
11:53:15 | INFO     | config.settings:write | demo.tasks.generate_report: 2df107ac-db09-4aa1-bed1-075544a05107 executed in 3.000s
11:53:15 | INFO     | demo.tasks:generate_report | Report weekly generated successfully
11:53:15 | INFO     | config.settings:write | demo.tasks.generate_report: 534c6768-38b8-4847-a2fd-f0dd963eb8c3 executed in 3.000s
```

That's it, problem solved. Your processes all run, you can switch from one to another with keyboard shortcuts or mouse clicks, if one dies, you see its name turn red:



```
Browserslist: caniuse-lite is outdated. Please run:
  npx update-browserslist-db@latest
  Why you should do it regularly: https://github.com/browserslist/update-db#readme

Rebuilding...

Done in 151ms.
```

You can restart it manually with `r` or setup an autorestart policy.

Of course, you can couple it with the excellent just task runner. → https://www.bitecode.dev/p/justified to make things even easier:

```
procs:
  django: just runserver
  tailwind: just watch_css
  huey: just queue
```

That's going to be a short blog post for once, because that's really it (and I'm writing this in between eating too much sugar and too much fat). Sure, you can do a bit more in `mprocs` configuration files. `mprocs` lets you define the current working directory, env variables, or select commands depending on OSes, but since `just` does it has well, I'd rather configure it there.

Honestly, I'm even going to hide `mprocs` behind `just dev`. I want `just` to be my interface to everything, so that I have only one place to look at when I switch between projects.

It’s easier for the team, and it’s also more efficient for AI agents, especially since the latter would use `just` but not `mprocs`.

Another nice Rust CLI tool I added to my `PATH`. There is a lot of that going on these days.

Jon's Arm Reference

Source: https://arm.jonpalmisc.com/

This site offers reference documentation for the AArch64 instruction set and system registers defined by the Armv8-A and Armv9-A architectures.

You can use the search field below (and at the top of every page) to search for an instruction or system register. Alternatively, you can browse the full listings manually:

- All Instructions: Base → https://arm.jonpalmisc.com/latest_aarch64/index.html | SIMD → https://arm.jonpalmisc.com/latest_aarch64/fpsimdindex.html | SVE → https://arm.jonpalmisc.com/latest_aarch64/sveindex.html | SME → https://arm.jonpalmisc.com/latest_aarch64/mortlachindex.html
- All Registers: AArch64 → https://arm.jonpalmisc.com/latest_sysreg/AArch64-regindex.html | AArch32 → https://arm.jonpalmisc.com/latest_sysreg/AArch32-regindex.html

If you find this site useful or have any suggestions, don't hesitate to let me know → https://jonpalmisc.com/contact.
If the search field above isn't working, your browser either does not support WebAssembly → https://webassembly.org/ or it has been disabled, e.g. in Apple's Lockdown Mode → https://support.apple.com/en-us/105120.

Optimize for momentum

By Murat
Published Dec 2, 2025, 8:21 PM
Source: https://muratbuffalo.blogspot.com/2025/12/optimize-for-momentum.html?m=1

Progress comes from motion. Momentum is the invisible engine of any significant work. A project feels daunting when you face it as a blank page. It feels easier when you built some momentum with some next steps. So, momentum makes the difference between blocked and flowing.

Think of a stalled truck on a desert road. You can't lift it with superhuman strength. But by rocking it with small periodic forces at the right rhythm (matching its natural frequency) you can get it rolling. Each tiny push adds to the previous one because the timing aligns with the system's response. The truck starts to move, and then the engine catches.

Projects behave the same way. A big project has its own rhythm. If you revisit it daily, even briefly, your pushes line up. Your brain stays warm. Context stays loaded → https://muratbuffalo.blogspot.com/2021/12/learning-technical-subject.html. Ideas from yesterday are still alive today. Each session amplifies the last because you are operating in phase with your own momentum. When you produce something every day, → https://muratbuffalo.blogspot.com/2024/07/advice-to-young.html you never feel stuck. You end each session with a clear record of progress. A researcher who touches their project daily is always a day away from a breakthrough. Skip too many days and you fall out of resonance. Then the project feels heavy again and needs a large effort to budge.

So the trick is to design your workflow around staying in motion. Don't wait for the perfect idea or the right mood. Act first. Clarity comes after. If a task feels intimidating, cut it down until it becomes trivial. Open the file. Draft one paragraph. Try freewriting → https://muratbuffalo.blogspot.com/2018/05/book-review-accidental-genius-using.html. Run one experiment. Or sketch one figure. You want the smallest possible task that gets the wheel turning. Completion, even on tiny tasks, builds momentum and creates the energy to do the next thing. The goal is to get traction and stop getting blocked on an empty page.

A messy page is better than an empty page to get started. In an interesting Machintosh folklore story → https://folklore.org/Make_a_Mess,_Clean_it_Up!.html, Burrell Smith deliberately made a mess in the classic video game Defender. He shot his own humans and let all mutants loose, just so he could figure out how to clean up the chaos wholesale. Fire and maneuver!

Practical tips

This is where I find LLMs help tremendously. (Haha. AI butts its head even in an advice column.) When you face a large messy problem, ask the model to break it into a sequence of concrete subtasks: "List the next ten actions for the experiment" or "Suggest a structure for this section". Then ask the LLM to do one of the easiest tasks in this list. The mediocrity will annoy you just enough to fix it. And now you are moving. We are getting somewhere.

A ten-minute timer → https://muratbuffalo.blogspot.com/2016/01/my-new-pomodoro-workflow.html is one of the simplest ways to get things going. Ten minutes is short enough that you can do almost anything for ten minutes. Pick a tiny task, and start. Most of the time you keep going after the timer ends because starting was the hard part. The timer lowers the activation energy and creates the first push on the flywheel.

Another way to build momentum is to work on the part of the project that feels most attractive at the moment. If you are not in the mood to write the introduction but feel curious about running a side experiment, do the experiment. If you feel more like drawing a diagram, draw it. Interest/love/curiosity is your fuel. Progress is progress. Nobody hands out medals for following a linear plan. The only requirement is that you keep adding small meaningful pieces to the project.

Momentum is not a glamorous, sexy idea. But it is reliable. Think of your work as a flywheel → https://en.wikipedia.org/wiki/Flywheel. Each nudge adds speed, and over the weeks, the wheel becomes powerful and unstoppable. People often admire the end state but they don't see the messy daily pushes that built the momentum.

why i think jj-vcs is worth your time

By steveklabnik
Source: https://lobste.rs/s/nrjotv/why_i_think_jj_vcs_is_worth_your_time#c_9ozl9l

You are missing a bit, though also, it's totally fine to just not like jj, so it might not be for you. But here's what I'd share about what you've written:

> • commit has a change id that is stable - so what? I don't see why this matters

Full agree. It alone isn't a feature. But it is something that enables other features. So it's not particularly compelling as its own thing, it's just a property of the system that's worth knowing about since you'll interact with it.

> • no stash, just jj new - not really sure why this matters, I've never had an issue with wip commits or stashes

It matters in the sense that you can get more things done with fewer concepts, which makes learning and usage easier. I often used wip commits in git because the stacked nature of the stash wasn't helpful, and that it was outside of the normal system of everything made it less powerful and easy to forget the details of.

jj's approach is like wip commits, but even easier. This stuff isn't about being life changing so much as just a bit nicer.

> • don't have to name branches - I like names, but feel like auto-naming could just be a git feature and not a reason to change my entire toolchain

It's not really "auto-naming", it's that you don't need a name at all. git calls this a "detached head." This is the mechanism that makes "you don't need a separate stash command" work, because you just create a new change on top of the one you're on. Git would force you to put a name here, but there's no real reason you should have to, it's just how git was designed and works.

Again, it's not that this is some sort of "holy shit" moment, it's just a nicer and simpler way to work. Name things when you want to give them names, don't when you don't.

> • it’s easy to work on top of a merge of a bunch of branches all at once - I don't know what this means, but it sounds confusing and I've never wanted to do this

For people with a lot of outstanding feature work, it can be nice to work in a way where all of those outstanding changes are still in your tree, so that you can build on top of them while you're waiting for them to be integrated. That's what this means, and if you're not in that situation, then it isn't particularly useful. I basically never work in this way, for example.

> • stuff gets auto-added - I think this is an anti-feature. Every VCS I've ever used required you to explicitly add stuff. This seems extremely good, so I would probably disable this feature of jj

This is a very reasonable reaction, but there's some subtleties you're missing. In git, you commit once you're done with work. With jj, you sculpt changes into what you want. So you can gain back the "I explicitly add stuff" via a workflow, even if in a literal sense jj adds everything into a commit. Incidentally, that workflow is one of the most popular ways of working with jj, because clean commit histories are good. It's called the "squash workflow" and I can elaborate if you'd like but there's already a lot here so I'll leave it at that for now :)

> • rebasing is error prone - in my experience, it's fine. I will pull from main, rebase my branch off of main, then keep working. I will interactive rebase if I need to change or re-order things. It works well.

For sure, I never had a particular problem with rebase either. What I will say is that I like jj because it makes using rebase really nice. Even if you don't have a problem with rebases in git, that also doesn't mean that there's no room for improvements: jj's approach to conflicts here is straight up better, in my opinion.

> Another negative is that since there is no JJHub, you likely need to work with GitHub, and the ux around that is fairly clunky. It's great that it works, but it's cumbersome due to some conceptual mismatches between the two tools.

I agree that a jj-native forge would be nicer to use. Tangled is good for some use cases but there's room for more here too.