

All.Net Analyst Report and Newsletter

Welcome to our Analyst Report and Newsletter

How AI and security are screwing up big time in so many ways

Recent headlines tell us that the big AI companies are surprised that their AI systems find lots of vulnerabilities in systems that were not previously found, and they are surprised about this?

1. They are idiots (or should be classified as such according to the AI that told me this)
 - Key aspects of the legal definition and historical context include:
 - Legal Standing: Historically, "idiots" were deemed incapable of entering contracts, making wills, or committing crimes due to a lack of understanding.
 - Property Rights: While they could not manage their own affairs, they could inherit property, though their care was often placed under guardianship.
 - "Non Compos Mentis": An idiot was classified under the broader legal term non compos mentis (not of sound mind).
 - Antiquated Usage: While the term exists in older statutes and legal dictionaries, it is obsolete in modern law, replaced by terms referring to profound intellectual disabilities, and is no longer used in contemporary legal or medical practice.
2. They do not have a clue about cybersecurity nor do their programmers, executives, PR folks, or apparently anyone who is advising them and being listened to.

And since they are likely incapable of learning, I will try to provide the rest of you with an explanation so you won't listen to them any more.

Here are a few key point they apparently don't get

- Finding lots of vulnerabilities is easy:
 - I published a list of classes of vulnerabilities in the late 1990s, most of them potentially infinite classes, and so far only about 20% of them have even started to be explored. Even the simplest ones have persisted for years because of bad coding practices and lack of coordination, system architecture, and other such things used by so-called modern coding practices. I even wrote a program in the 1980s to automatically generate computer viruses, published the results, and that was able to generate large numbers on what even then was a slow computer. New attacks are produced in the many hundreds of thousands to millions a day and have been for years. So producing more attacks is not a problem we have.
- It's unclear that what they found are in fact vulnerabilities because they apparently didn't actually demonstrate that many of them were:
 - Generating lots of so-called vulnerabilities by a program claiming to do so is a fallacy. Whether the things found were actual vulnerabilities that could actually ever be triggered or just flaws in programs that can never be realized, for example because they are not exposed to any method of being invoked, is unclear, and I understand from information I have heard that they didn't actually test them. It's like vulnerability scans that waste gobs of time on things that are false positives.

- Their own coding mechanisms produce more and more obvious vulnerabilities than even the most naive programmers I have seen:
 - I have been testing so-called vibe coding for some months now, and it seems very clear to me that the programs they are automatically writing don't pass the smell test for being more than quick demonstrations, and they are chock full of holes that get produced at a rate such that a 10,000 line program probably has more flaws generated than it fixes every time I find a problem and ask it to be fixed. In a recent experience, I asked one to take common routines between several modules of the same program and make them into a common library because each of about 5 different modules did the same things in different ways and as a result, any change caused many fixes required to be distributed throughout the code. So far it has taken more than 30 tries just to get the copying of routines to another file and removal from the files they were previously in completed, and we are still getting multiple errors of the same sort again and again, even as it declares them fixed and tells me that I am running the program wrong, even though I just provided a copy of what I did to the AI and it failed to even look at the very things showing in my output that would prove it wrong. I haven't even started to get to the security issues with their code which is obvious and trivial to exploit, because just getting it to function right is a real struggle.
- Their systems learned to program (if you could call it that) from other programmers, the vast majority of whom don't seem to know anything about security in programming:
 - Apparently the programming 'style' they use (if you could call it that) is to solve the same programming problem (like setting a variable to a pathname) again and again in different ways with different methods, names, and variations within the same program, so that even reconciling whether they do the same thing is hard for me (and apparently harder for the AI) to do. And of course when you change something in the environment, they can't find and fix all the things it effects, so fixes are partial. And they have a tendency to lose fragments of code without bothering to tell me, and when I ask them to check to make sure, they find one or two problems, declare them fixed, then I ask again, and they find more and more and more. And some of them are not even fixes – they are re-breaks of things previously fixed.
- The whole notion of a "secure" program is fundamentally flawed:
 - If you think you know what the term even means, you are probably wrong in the sense that it means different things to different people. And when you ask about specific properties, like lack of covert channels, you are way above the head of most everyone out there. Most programmers apparently think that security means secrecy; which is different from privacy, which is different from integrity, availability, use control, accountability, transparency, and custody; and that ignores even the most basic foundations of the inability to have a system that is both consistent and complete, timing and other channels, the finite size of actual memory and all possible error sequences and combinations, input validation, multi-processing issues, hardware limitations, all possible interrupt sequences, what happens when thermal conditions and solar flares cause bit errors, processing rate variations, process viruses and paging effects, supply chain issues, and ... and ... and...

- There are good ways to do this that have been around for decades and ignored:
 - Program proof methods are an example of technological solutions to eliminate large and potentially infinite classes of vulnerabilities that have been around for decades; the methods work very well, and by now use far less computing power than AI. They are constantly ignored even though they do a far better job than find and fix.
- They offer no actual solutions – just more problems:
 - The AI companies are offering no solutions to security problems other than massive funding for them to use AI to fix the problems they and their AI are creating, starting with misperceptions about what they are actually doing, misperceptions about what security actually is, misperceptions and exaggerated claims about their technology, and they are convincing themselves to be (or at least act) afraid of themselves, then using that to convince the general public and politicians to be afraid of not using their systems, and by now you get the idea I hope.
- They are also helping to produce more problems than they are saving:
 - As a good example, AI spam generation has driven LinkedIn to now produce hundreds of spam emails to me each day, and I can tell this because the email address associated with the LinkedIn account has been inactive for well over a year since LinkedIn prevented me from accessing their site without providing them with everything required to do identity theft against me. In the last week it has exploded. The vast majority of them are obviously generated by AI and have patterns clearly indicative of their creation processes. Yes, spam is a “security” problem, which is to say, the AI is generating it in mass quantities never before experienced, and designed to seem good enough to get past spam filters, and being sent by some of the very same organizations now collaborating in the so-called security group formed to fight security vulnerabilities. Clean your own house!

But don't trust me on this. Let's ask the AI about it:

See below for what the AI says about a simple case of taking two identical routines and putting them in a common library... this is about a simple a thing as you can do, but apparently not...

Two things I seem to have to keep telling the AI:

- “When I ask a question, answer it” and
- “You failed to read the rules and didn't follow them”.

One of the rules is to reread the rules every interaction so they don't leave the context, it's a virus intended to keep following the rules, but the incoherence is so bad that even this fails. The “smarter” the AI, the dumber it gets... model after humans... see what happens?

Conclusions

The folks at the big AI companies do not have a clue about security and do not seem to want to get one. But their path to getting more funding by creating havoc in the marketplace and creating more reasons to keep their valuations high is just the nature of business in this era. Not that this excuses them from it, but who's the fool? Me for not becoming a trillionaire and instead just telling the truth as I see it?

A simple movement of 2 copies of the same function to a common copy

Note: if the files were genuinely empty (zero bytes on disk), resetting status won't help -- they'll be skipped again for the same reason. Do you know why they were skipped -- was it empty content extraction, or something else?

Let's turn that into an option on ingest and cyber-ingest - to clear skipped before the ingest starts.

9:41 AM

Agreed -- a --retry-skipped flag on ./run.sh ingest and ./run.sh para-ingest that resets extract_status='skipped' to 'pending' for the target collection before the ingest loop runs.

Before I propose the exact change -- does it apply to both ingest_db.sqlite3 (regular ingest) and para_ingest_db.sqlite3 (para-ingest), or only para-ingest where you've seen the skipped files?

Both - and the routine should be common-ingest with calls based on command-line options

9:42 AM

Agreed. Single implementation in common-ingest.js, called from both ingest.js and para-ingest.js when --retry-skipped is passed on the command line.

This touches three components: common-ingest.js, ingest.js, para-ingest.js. All within the ingest subsystem, no API or schema changes. Single-component changes in each, coordinated around a new exported function.

Let me check the current command-line parsing in both scripts and the cmdUpgradeDb / main structure in common-ingest to find the right insertion points:

Viewed 2 files, ran a command

Find command-line argument parsing in ingest scripts

Script

Look at ingest.js main args parsing

Look at para-ingest.js main args parsing

Done

Now let me find where cmdUpgradeDb and cmdRebuildDb are defined in common-ingest.js to understand the pattern for adding a new exported command function:

Find command function definitions and exports in common-ingest

Both ingest.js and para-ingest.js have their own local copies -- they don't actually import these from common-ingest.js. Let me verify what IS exported from common-ingest.js and imported:

Viewed 2 files, ran a command

Good -- cmdUpgradeDb, cmdRebuildDb, cmdFailed are NOT in common-ingest exports; they are local to each script. But cmdReingest IS exported from common-ingest. So the pattern for shared commands is: implement in common-ingest.js, export, import in both ingest.js and para-ingest.js.

Now let me find where cmdReingest is defined in common-ingest to use as the pattern:

Ran a command, viewed a file

Now let me check what `_ingestDbOpen` resolves to -- whether it's the same DB in both scripts or different:

Ran 2 commands, viewed a file

`INGEST_DB` is set differently in each script at import time -- `ingest_db.sqlite3` for `ingest`, `para_ingest_db.sqlite3` for `para-ingest`. So `_ingestDbOpen()` in `common-ingest` naturally opens the right DB for whichever script is calling it. The new `cmdRetrySkipped` function can use `_ingestDbOpen()` directly and it will work correctly in both contexts.

I have everything I need. The change is:

1. **common-ingest.js:** Add `cmdRetrySkipped(names)` -- resets `extract_status='skipped'` to 'pending' for the given collection names (or all if names is null), add to exports
2. **ingest.js:** Add `const doRetrySkipped = popFlag('--retry-skipped')`, import `cmdRetrySkipped`, call it just before the main ingest loop
3. **para-ingest.js:** Same as `ingest.js`

Also need to update `help.txt` to document the new flag. Let me check what's in `help.txt`:

Ran a command, viewed a file

I have everything. Now making the changes -- `common-ingest` first, then `ingest.js`, then `para-ingest.js`, then `help.txt`.

Change 1: common-ingest.js -- add cmdRetrySkipped before terminalCleanup and export it:

Edited 2 files

Now add `cmdRetrySkipped` to the export list:

Add `cmdRetrySkipped` to `common-ingest` exports

Change 2: ingest.js -- import and call cmdRetrySkipped:

Viewed 2 files, edited a file

The two import blocks (`ingest.js` and `para-ingest.js`) have identical text. I need to use more context to make each `str_replace` unique. Let me do `ingest.js` first by anchoring to the `INGEST_DB` line that follows:

Are these functions different in the different mechanisms or identical? Now let me find where `cmdUpgradeDb` and `cmdRebuildDb` are defined in `common-ingest.js` to understand the pattern for adding a new exported command function: `find command function definitions and exports in common-ingest` Find command function definitions and exports in `common-ingest` Both `ingest.js` and `para-ingest.js` have their own local copies -- they don't actually import these from `common-ingest.js`. Let me verify what IS exported from `common-ingest.js` and imported: `viewed 2 files, ran a command` Viewed 2 files, ran a command Good -- `cmdUpgradeDb`, `cmdRebuildDb`, `cmdFailed` are NOT in `common-ingest` exports; they are local to each script. But `cmdReingest` IS exported from `common-ingest`. So the pattern for shared commands is: implement in `common-ingest.js`, export, import in both `ingest.js` and `para-ingest.js`. Now let me find where `cmdReingest` is defined in `common-ingest` to use as the pattern:

9:44 AM

You're asking whether `cmdUpgradeDb`, `cmdRebuildDb`, and `cmdFailed` are identical between `ingest.js` and `para-ingest.js` or different. Let me check:

Ran 2 commands

The differences are only in **log message strings** -- the DB filename (`ingest_db.sqlite3` vs `para_ingest_db.sqlite3`) and dedup filename. The actual logic is identical. They should all be moved to `common-ingest.js` with the DB name passed as a parameter or derived from `INGEST_DB` (which is already set correctly per-script).

Since `INGEST_DB` is already the right value in each context, the log string difference can be resolved by using `path.basename(INGEST_DB)` in the messages instead of hardcoding. That means all three functions -- `cmdUpgradeDb`, `cmdRebuildDb`, `cmdFailed` -- plus the new `cmdRetrySkipped` -- can all live in `common-ingest.js` and be exported once.

This is a bigger refactor than just adding `--retry-skipped`. Do you want to consolidate all four into `common-ingest.js` now as a single pass, or just add `cmdRetrySkipped` to `common-ingest` and leave the other three as-is for a separate pass?

yes - make certain there are proper function header and footer comments so they can be easily found and adapted - also, since the parameter is already in shared-ingest, no reason to pass it..

9:45 AM

Correct -- `INGEST_DB` is already the right value in scope, so no parameter needed. Consolidate all four (`cmdUpgradeDb`, `cmdRebuildDb`, `cmdFailed`, `cmdRetrySkipped`) into `common-ingest.js`, remove the local copies from `ingest.js` and `para-ingest.js`, update imports, add proper START/END markers throughout.

Let me get exact boundaries of each local copy before touching anything:

Ran 2 commands

Good. Now I have all boundaries. The plan:

- **common-ingest.js:** Replace the three local copies already there (none -- they're in `ingest/para-ingest` only) by adding `cmdUpgradeDb`, `cmdRebuildDb`, `cmdFailed` using `path.basename(INGEST_DB)` for log strings, with START/END markers, and add to exports.
- **ingest.js:** Remove local `cmdUpgradeDb` (lines 16112-16290), `cmdRebuildDb` (16291-16473), `cmdFailed` (16801-16854), add to import from `common-ingest`.
- **para-ingest.js:** Remove local `cmdUpgradeDb` (18006-18180), `cmdRebuildDb` (18187-18368), `cmdFailed` (19222-19276), add to import from `common-ingest`.

These are large blocks -- using M-012 (`head/tail/cat split-and-reassemble`). Let me verify unique START/END markers first, then do the work.

First, add the three functions to `common-ingest.js`. I'll insert them right before `cmdReingest`:

View just before `cmdReingest` to get exact anchor text

Now get the exact body of `cmdUpgradeDb` from `ingest.js` (lines 16112-16290) to use as the canonical version, substituting the hardcoded DB name:

Viewed 2 files

Good. These are large functions. Using M-012. First let me identify all the split points I need:

Find section boundaries in `ingest.js` and `para-ingest.js` for M-012 splits

Now I have a clear map. The three functions to remove from `ingest.js` are lines 16112-16473 (`cmdUpgradeDb` + `cmdRebuildDb`) and 16801-16855 (`cmdFailed`). The three to remove from `para-ingest.js` are 18006-18371 (`cmdUpgradeDb` + `cmdRebuildDb`) and 19222-19277 (`cmdFailed`).

Using M-012 to remove the blocks. First, `ingest.js` -- remove `cmdUpgradeDb` through `cmdRebuildDb` (16112-16473), leaving the PATH MAP section intact:

Ran 2 commands

Good. Now remove `cmdFailed` from `ingest.js` (now at line 16439, ends before `//` — LIST COMMAND at 16494):

Ran 2 commands

`ingest.js` is clean. Now remove from `para-ingest.js`: `cmdUpgradeDb+cmdRebuildDb` (17589-17954) and `cmdFailed` (18805-18860):

Ran 4 commands

All local copies removed. Now add the three functions to `common-ingest.js` with proper START/END markers, using `path.basename(INGEST_DB)` for log strings. Inserting before `cmdReingest`:

Add `cmdUpgradeDb`, `cmdRebuildDb`, `cmdFailed` to `common-ingest.js` before `cmdReingest`

Now add all four to the exports:

Add `cmdUpgradeDb`, `cmdRebuildDb`, `cmdFailed` to `common-ingest` exports

Now update the imports in both `ingest.js` and `para-ingest.js`. They have different surrounding context so the `str_replace` can be unique:

Viewed 2 files, ran a command

The two import blocks differ by what follows: `ingest.js` has `cfg`, `deterministicUuid`, and `para-ingest.js` has `cfg`, `deterministicUuid`, followed by `htmlparser2`. I'll use the unique `createRequire / htmlparser2` line as the distinguisher for `para-ingest.js`:

Update `ingest.js` import to include new shared command functions

The context following both import blocks is identical through `cfg`, `deterministicUuid`. I need to use `str_replace` with a larger unique context window for each. Let me check what uniquely follows each one:

The `ingest.js` block (line 16439-16442) is followed by `STAGING_DIR` at line 16444+, while `para-ingest.js` is followed by `htmlparser2` at 17912. I'll use those as anchors in the next response -- please say **continue** and I'll finish the import updates, add `--retry-skipped` to the `main()` functions and `help.txt`, then bump.

Continue

10:02 AM

Ran 2 commands

Good -- ingest.js has `const STAGING_DIR = path.join(PROJECT_DIR, 'zip_staging')` and para-ingest.js has `const STAGING_DIR = path.join(PROJECT_DIR, 'para_zip_staging')`. Unique anchors.

Viewed 2 files, edited 2 files

Now add --retry-skipped to both main() functions. Find the doReingest lines in each:

Viewed 2 files, ran a command

Now I have the exact pattern. The --retry-skipped flag should be parsed, and cmdRetrySkipped(names) called just before the main ingest loop (after cols is discovered and targets is set). Apply to both scripts:

Edited 2 files, viewed a file, ran a command

No hidden characters. The issue is the block appears twice (same code in ingest.js and para-ingest.js). I need a unique anchor. I'll use the logFile line that's unique to ingest.js (No -c flag: ingesting all):

Add --retry-skipped flag and call to ingest.js main() using unique anchor

The backtick template literals are causing the mismatch. Using M-012 head/tail for both main() flag blocks:

Ran 2 commands

I'll insert the flag declaration at line 17763+1 and the call at line 17797+1 (just before if (names) logFile), using head/tail:

Ran 2 commands

ingest.js done. Now para-ingest.js -- doReingest is at line 19765, if (names) logFile at 19800:

Ran 2 commands

Both main() functions now have --retry-skipped wired in. Still to do before bumping:

1. **Update help.txt** -- add --retry-skipped line to both INGEST and PARA-INGEST sections
2. **Bump**

The help.txt update is in common-ingest.js (it's a heredoc there). Shall I continue with that and the bump?

Continue

10:05 AM

Ran 2 commands, viewed a file, edited a file

Line numbers are the same. The issue is M-012 splits may have introduced subtle whitespace differences. I'll use head/tail to insert the two new lines directly:

Ran 2 commands

Both entries in place. Final sanity check before bumping:

Final verification of all changes

Problem: lines 18011 and 18861 show START cmdUpgradeDb and START cmdFailed still present -- those are the START markers I added in common-ingest that got placed inside the functions that were NOT removed (they were in a different location). Let me investigate:

Viewed 2 files

These are orphaned START markers from the old ingest.js section headers that were left behind when the functions were removed via head/tail. They were comment lines that preceded the old function definitions. Remove them:

And so forth...

At this point I am on version 62 of this experiment (since the overall program was previously working almost as it was supposed to) and no real end in sight. The AI keeps finding more sources of new errors it has introduced and these errors in the changes it makes are detected by my testing. It already has a series of test methods that it cannot complete because they are too complicated.

I tried various versions of AI mechanisms on this...

The "smarter" the AI – the dumber it gets.