White Paper

NETRISE

Your Security Scans Are Missing Critical Vulnerabilities

Here's why. . .

If you're responsible for software in any capacity—whether you're writing it, securing it, or managing it—your current vulnerability scanners are likely missing significant portions of your actual attack surface.

I've spent the last year and a half watching our binary analysis platform reveal vulnerabilities that conventional scanners completely miss. And honestly? Each discovery is eye-opening for everyone involved—including us. Every instance reinforces the same lesson: the code you think you're running isn't the code that's actually running.

The Team Behind These Insights

As VP of Engineering, I spend more time in architecture discussions and strategy meetings than deep in the code these days. But here's what gives me the confidence to write about binary analysis: I work directly with some of the world's leading experts in this field.

Anthony Feddersen
VP of Engineering, Netrise

Our team includes Craig Heffner, creator of binwalk and former NSA Red Team member; Peter Eacmen and Terry Dunlap, both former cofounders of RefirmLabs with extensive experience across intelligence agencies and military branches; and our CTO, Michael Scott, formerly at the Marine Corps Cyber Warfare Group within USCYBERCOMMAND.

Working alongside these experts for years has given me a unique vantage point – I've learned from their deep technical expertise while also seeing how these concepts apply to broader engineering and business challenges. They've reviewed every technical detail in this post, ensuring accuracy while I focus on making these insights accessible and actionable.

Let me show you what we've found.

Real World Case Studies

When OpenSSL 3.0.0 Hides in Plain Sight

When the NetRise Platform® flagged `OpenSSL 3.0.0` in a firmware image for a widely used network operating system, the device owner escalated immediately. They had every right to be skeptical—their own system diagnostics showed a completely different, and seemingly safe, story. For this specific device, the installed package manager reported `OpenSSL 3.0.7`:

```
$ rpm -q openssl
openssl-3.0.7-41869325.[redacted].1.x86_64
$ ls /lib64/libssl.so.*
/lib64/libssl.so.3.2.2
```

We take this kind of escalation very seriously at NetRise. When customers challenge our findings, we don't just point to automated results—we break out our manual analysis tools and leverage years of binary forensics expertise to provide concrete evidence. In this case, we went straight to the source and examined the Python SSL modules directly:

```
$ strings /usr/lib64/python3.9/lib-dynload/_ssl.cpython-39-x86_64-linux-gnu.so | grep “OPENSSL_”
OPENSSL_3.0.0
OPENSSL_sk_new_null
OPENSSL_sk_num
OPENSSL_sk_pop_free
```

There it was—`OpenSSL 3.0.0`, sitting inside Python's SSL module. Our manual investigation revealed it everywhere Python touched crypto:

```
Pattern ‘OPENSSL_3.0.0’ found in:
  - /usr/lib64/python2.7/lib-dynload/_ssl.so
  - /usr/lib64/python2.7/lib-dynload/_hashlib.so
  - /usr/lib64/python3.9/lib-dynload/_ssl.cpython-39-x86_64-linux-gnu.so
  - /usr/lib64/python3.9/lib-dynload/_hashlib.cpython-39-x86_64-linux-gnu.so
  - /usr/lib64/python3.9/site-packages/cryptography/hazmat/bindings/_openssl.abi3.so
  - /usr/lib64/python3.9/site-packages/M2Crypto/_m2crypto.cpython-39-x86_64-linux-gnu.so
  - /usr/lib64/python3.9/site-packages/pycurl.cpython-39-x86_64-linux-gnu.so
```

Here's what happened: When Python was compiled for this widely used network operating system, it was built against `OpenSSL 3.0.0`. The Python modules contain statically linked OpenSSL code that's frozen in time. System library updates don't touch it. This means vulnerabilities like CVE-2022-3602 and CVE-2022-3786 could still be exploitable through Python SSL operations, even though the system library shows as patched.

The customer's own diagnostic actually confirmed our finding:

```
$ ldd ./_hashlib.cpython-39-x86_64-linux-gnu.so
linux-vdso.so.1 (0x00007fff9384b000)
libcrypto.so.3 => /lib64/libcrypto.so.3 (0x00007f541a35b000)
```

While ldd shows dynamic linking to libcrypto.so.3 for some functions, the module also contains statically linked OpenSSL 3.0.0 code that ldd can't see. So your dependency scanning tools report that you're using the current OpenSSL version based on the dynamic links, but they're completely missing the vulnerable legacy code that's actually compiled directly into your binary. The tools show you one thing, but the reality of what's running in production is entirely different.

The Vendored Library Problem

Another escalation came from a manufacturer of IoT devices when we reported `zlib 1.2.8` in their embedded Linux system. They pushed back hard, insisting they ship `zlib 1.3.1`:

```
$ ls /lib/libz.so.*
/lib/libz.so.1.3.1
$ rpm -q zlib
zlib-1.3.1-1.el9.x86_64
```

Again — this type of escalation is not uncommon—customers see conflicting information between our platform results and their system queries, and they're right to question it. When NetRise identifies components that don't match package manager output, we'll dig deeper with our manual analysis toolkit to give customers the evidence they need if they're not satisfied with the evidence we've already provided in the platform. In this case, we traced the old `zlib` version to `/bin/rsync` and dove into the source code investigation that our platform had flagged.

Our manual verification revealed the culprit—`rsync` vendors their own copy of zlib:

```
rsync/
├── zlib/           # Vendored zlib 1.2.8
│   ├── deflate.c
│   ├── inflate.c
│   └── zlib.h      # Version: 1.2.8
├── configure.ac    # Line 162: --with-included-zlib option
└── Makefile.in
```

The rsync project includes `zlib 1.2.8` in their source tree by default. You can compile it to use the system zlib with `--with-included-zlib=no`, but most distributions use the default configuration. So every system shipping a standard rsync binary has this old zlib version embedded in it, regardless of what the package manager reports.

The Hidden Dependency Problem

When you run make or npm build, your compiler doesn't just translate source code. It resolves symbols, links libraries, performs dead code elimination, and applies optimizations. Each step involves critical choices: Should it use the system OpenSSL at `/usr/lib/libssl.so.3` or the one in your build cache at `/opt/build/deps/libssl`.so.1.1? When `gcc` encounters `-O2`, it might inline entire functions from static libraries. When Go compiles with `CGO_ENABLED=1`, it's making runtime versus compile-time linking decisions that completely change your attack surface.

The core problem is that vulnerable libraries get baked directly into your binaries through static linking and vendoring. Meanwhile, your conventional scanners—the ones checking `package.json`, `go.mod`, or `pom.xml`—are analyzing intention, not reality. They have no visibility into what actually happened during compilation.

Compilation is a complex, multi-stage process where your compiler makes thousands of decisions that fundamentally shape your final binary—and each decision can introduce security risks that your source-code scanning tools can't see.

Here's how compilation introduces vulnerabilities that manifest-based scanning will never catch:

Dependency Resolution at Build Time

Your build system makes real-time decisions. Take Maven's nearest-wins strategy—if two dependencies require different versions of commons-logging, Maven picks the one closest to your root `pom.xml`. But here's what happens: your CI environment has `commons-logging 1.1.1` in its `.m2` cache from a previous build. Your manifest declares `1.2`, but Maven finds `1.1.1` first and uses it. The manifest scanner reports `1.2`. Your binary contains `1.1.1` with CVE-2018-11771. We see this constantly with npm's hoisting behavior and Go's minimal version selection.

Transitive Dependencies Getting Compiled In

When you import `requests` in Python, it pulls in `urllib3`, which pulls in `certifi`, which might pull in an old `ssl` module. During compilation with `python setup.py build_ext`, these get resolved based on your build environment's state. If you're using `pip install --target` or building wheels, these transitive dependencies become part of your distributed binary. They're not in your requirements.txt, so your scanner misses them entirely.

Static Linking Behavior

This one's everywhere. Build a Go binary with `go build -tags netgo`, and you're statically linking the entire net package. Compile Node.js modules with `node-gyp`, and OpenSSL gets embedded directly into your `.node` files. Even webpack, when you use `node: { crypto: true }`, can embed crypto libraries into your bundle. The command `ldd` won't even show these because they're not dynamically linked—they're literally part of your executable's `.text` segment.

Build-Time Code Generation

Frameworks, like gRPC, generate code during compilation. Run `protoc --go_out=plugins=grpc:. *.proto`, and you're creating new code based on your proto definitions. If there's a vulnerability in the generated marshaling code, you'll never find it by scanning your source. Same with Spring Boot's annotation processors or Lombok's compile-time code generation. The vulnerable code doesn't exist until `javac` runs.

Vendored Dependencies with Modifications

We've found this pattern repeatedly: teams vendor dependencies using `go mod vendor` or `npm pack`, then apply patches. During compilation with flags like `-mod=vendor` or `--prefer-offline`, these modified versions get compiled in. They don't match any known version in vulnerability databases. One customer had a vendored `zlib 1.2.8` with custom patches in their rsync binary. The patches actually introduced a new vulnerability, but no scanner could detect it because this version literally doesn't exist outside their build.

We need to stop pretending that manifest files represent reality. They represent intent. If you want to know what vulnerabilities are actually in your production systems, you need to analyze the compiled binaries themselves.

The takeaway? Modern compilation involves thousands of micro-decisions—symbol resolution, link order, optimization levels, build flags. Each decision can introduce vulnerable code that doesn't appear in any manifest. You're scanning the architectural blueprint while the actual building was constructed by a compiler making autonomous decisions based on whatever it found in the build environment at that exact moment.

Understanding the Hidden Attack Surface

When we analyzed the widely used network operating system, the scope became clear. We found 47 components with hidden dependencies—14 cases of static linking (mostly OpenSSL in Python modules), three vendored libraries, and 30 instances of build-time injection.

Traditional Software Composition Analysis tools parse manifest files like package.json, pom.xml, or RPM databases. But they're completely blind to several categories of dependencies.

Static linking freezes library code at compile time. Our Python OpenSSL case shows this perfectly—modules compiled against `OpenSSL 3.0.0` keep that code forever, even after system upgrades to 3.2.2. It's baked in, not referenced.

Vendored dependencies happen when projects include third-party libraries in their source tree. The `rsync` team vendors `zlib` to ensure compatibility. We also found vendored popt and stunnel libraries in other binaries. These never show up in dependency manifests because they're part of the `main` package.

Build environment dependencies sneak in during compilation. Container base images contribute system packages. Build toolchains embed library versions. Transitive dependencies add components that nobody explicitly declared. The final binary contains code from all these sources, but the manifest only knows about direct dependencies.

Compilation flags change everything. Building rsync with default settings yields a version with vendored `zlib 1.2.8`. Build it with `--with-included-zlib=no` and you get system `zlib`. There's no way to tell which path was taken just by looking at the binary name.

How Binary Composition Analysis Works

Binary Composition Analysis examines compiled binaries rather than declared dependencies. Think of it as forensics for executables.

The detection process combines multiple techniques. We use signature-based identification to find binary patterns unique to specific library versions. Symbol analysis examines exported and imported symbols. File carving recursively extracts embedded components. When possible, we cross-reference findings with source code to understand why a component is there.
Here's how we investigated `rsync`:

First, binary signature detection identified `zlib 1.2.8`. We traced it to `/bin/rsync`. Symbol verification confirmed embedded zlib code. Source validation showed the GitHub repository contains a zlib directory. The root cause? Vendored dependency in the `rsync` source. The fix? Recompile `rsync` with `--with-included-zlib=no`.

Want to Investigate This Yourself?

If you're determined to dig into these issues manually, it's possible—but it requires a significant investment in time and specialized expertise. The process below gives you a glimpse into the painstaking, multi-tool investigation required for a *single binary*. Now, imagine scaling this across your entire software portfolio.

Here's a look at the manual playbook our experts use when they need to go deep.

First, you need to dissect the binary itself. We'd start with a tool like `binwalk` (built by our own Craig Heffner) to see what's embedded inside.

Start with binwalk – it's your best friend for analyzing firmware and binaries to see what's actually embedded inside them:

```
# Basic analysis to see embedded files and signatures
binwalk /path/to/binary
# Extract embedded components for deeper inspection
binwalk -e /path/to/binary
# Look for specific signatures (like OpenSSL or zlib)
binwalk --signature /path/to/binary
```

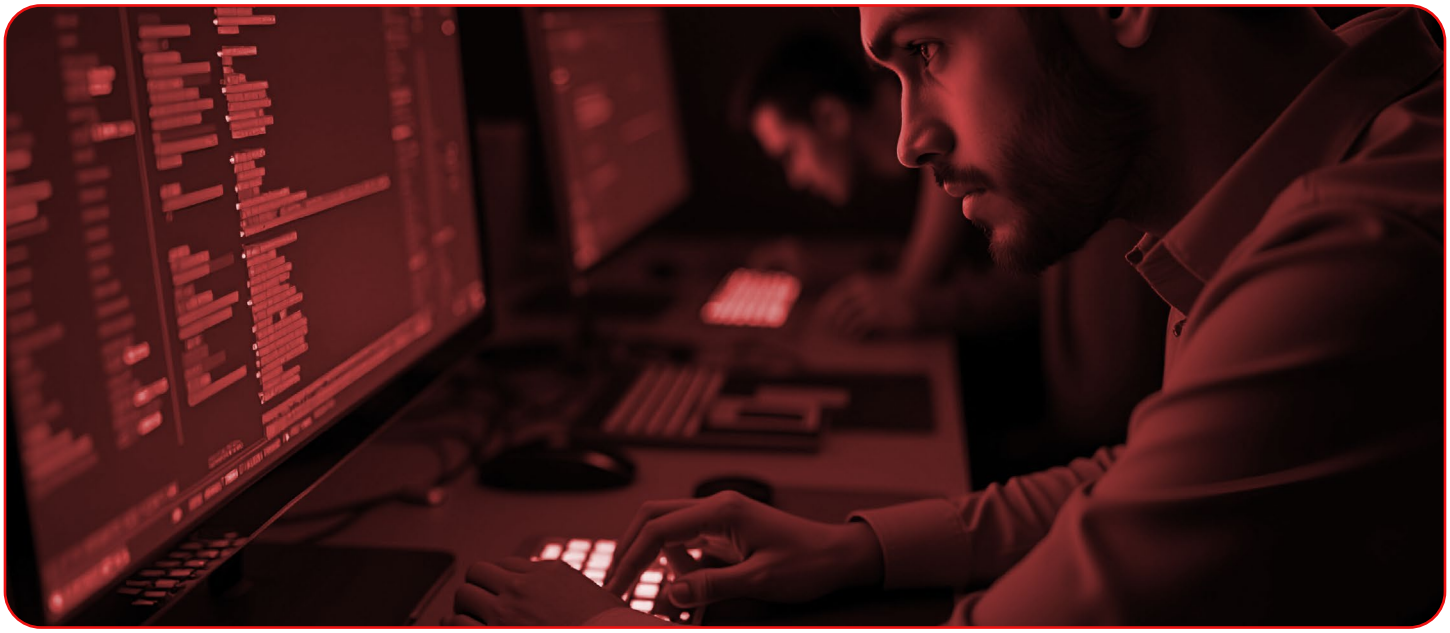
This is just the first step. Next, the real forensics begins. You'll need to hunt for clues like version strings, but grep is a blunt instrument. Sifting through the noise to find a real indicator requires a trained eye.

```
# Hunt for version strings in binaries
strings /path/to/binary | grep -E "OpenSSL [0-9]\.[0-9]\.[0-9]"
strings /path/to/binary | grep -E "zlib [0-9]\.[0-9]\.[0-9]"
# Examine symbols to see what functions are present
objdump -T /path/to/binary | grep OPENSSL
nm -D /path/to/binary | grep zlib
```

Finding a string or a symbol is a good start, but it isn't proof. You still need to dig into the source code and build configurations—becoming an archaeologist for build flags and vendored code. This is an often brittle, error-prone hunt.

```
# Look for vendored directories (they love to hide)
find . -type d \( -name "zlib" -o -name "openssl" -o -name "vendor" \)
# Examine build configuration for static linking flags
grep -r "with-included" configure.ac
grep -r "vendor" CMakeLists.txt
```

Pay special attention to build flags that affect how dependencies get compiled in. For rsync, watch for `--with-included-zlib=no` and `--with-included-popt=no`. Python builds use `--with-system-ffi` and `--with-system-expat`. And OpenSSL linking behavior changes completely depending on whether you're using `--enable-shared` versus static linking choices.



As you can see, this is a complex, expert-driven process for just one component. A proper "build-it-yourself" approach requires stitching together a dozen open-source tools (e.g., `binwalk, syft, grype`, etc.) into a custom pipeline. You have to hire the right people to build it, operate it, and—most importantly—interpret the fragmented results. Even then, you'll only get about 50% of the way to a comprehensive solution.

For most organizations, the investment in building and maintaining such a system, and hiring the scarce talent to run it, is simply unsustainable. The real challenge isn't just finding a single vulnerability; it's creating a scalable, automated program that provides a single source of truth for your entire attack surface. That's precisely the problem we built NetRise to solve.

Finding these hidden dependencies is just the beginning. Here's your action plan.

If You're on the Security Team

Don't dismiss unexpected findings—investigate them. Our rsync "false positive" turned out to be a vendored dependency the vendor didn't even know about. When your scanner finds something weird, that's usually where the interesting stuff is.

Document your validation process thoroughly, as you'll need it repeatedly. Begin by identifying the binary location, then check for vendored directories in the source. Next, verify the findings using `strings` or binary analysis, and finally, document which compilation flags impact the behavior. Build yourself a database of known vendoring patterns. We've found rsync vendors `zlib` and `popt`, Python can statically link `OpenSSL`, and there are plenty more out there.

If You're on the Development Team

Stop assuming your dependencies use system libraries. Run a quick check for vendored code in your upstream dependencies:

```
find . -type d -name "zlib" -o -name "openssl" -o -name "vendor"
```

Get familiar with your build flags. Know that rsync needs --with-included-zlib=no to use system zlib. Python needs --with-system-ffi --with-system-expat. Document all the flags that affect dependencies in your own software.
And here's the big one: when you patch a library, rebuild everything that might include it. Not just the direct dependencies—everything. Yes, it's a pain. Yes, it's necessary.

If You're a Vendor or in Leadership

You need to invest in real investigation capabilities. Our rsync investigation required binary analysis tools, source code examination, build system understanding, and cross-reference validation. This isn't something you can hand-wave away.

Make all of this visible in your SBOMs. Don't just list "zlib 1.3.1 (system library)." Show that rsync has zlib 1.2.8 vendored, note that it was built with default flags (vendored enabled), and document that recompilation with proper flags could fix it.

Time for some tough love: you don't actually know what you're shipping. Our investigations consistently show vendors being surprised by what's in their products.

The Bottom Line

The evidence from our investigations is undeniable. Vendors don't know what's in their products. Package managers miss vendored and statically linked code. Binary signatures reveal what's actually running. And when we investigate, we consistently confirm these findings. This problem is widespread and systematic, likely affecting you right now.

As one of our engineers put it: "This shows that code from a version of zlib that the system maker wasn't aware of [is] on their image due to an upstream package dependency vendoring." That's not a bug—it's how modern software development works.

This isn't about pointing fingers at specific vendors. It's an industry-wide problem. Every compiled binary potentially carries hidden dependencies. Every vendored library is a security update that won't reach its target. Every static link is a future CVE waiting to happen.

Binary Composition Analysis isn't just another security tool to add to your stack. It's the investigative process that reveals what's actually running in your environment versus what you think is running. And trust me, those are two very different things.

You've got hidden dependencies. We've proven it over and over. The question isn't whether they exist—it's whether you're going to find them before someone with less friendly intentions does.

Anthony Feddersen is the VP of Engineering at NetRise, where he leads the development of advanced binary analysis solutions to secure the software supply chain. His career has focused on building and securing foundational cloud services at scale, previously serving as head of engineering at Chainguard and in senior leadership roles at Electronic Arts, VMware, and Google.

NETRISE

What's Inside *Your* Software?

Get a Demo