PIPpro User Guide

Python IP Protection -- Source-Level Obfuscation & String Encryption

Version	2.0.0
Date	April 2026
Product	PIPpro by DenseDefense
Platforms	Windows (.exe installer) + Linux (.deb package)
Dependencies	Python 3.10+ (stdlib only for obfuscation)
Publisher	DenseDefense -- DFW, Texas
Website	densedefense.com

Table of Contents

1. Introduction

PIPpro (Python IP Protection) is a source-level Python obfuscator that protects intellectual property without requiring external tools, commercial licenses, or runtime dependencies. It operates directly on Python source code using the AST (Abstract Syntax Tree) module from Python's standard library, applying a multi-phase pipeline of transformations that make reverse engineering impractical.

PIPpro v2.0.0 is a HallMonitor-hardened release that addresses every security finding from the v1.x audit. The encryption engine has been upgraded from simple XOR to a SHA-256 key-derived stream cipher with HMAC-SHA256 integrity verification. Name mangling now covers ALL user-defined names by default, and two new obfuscation phases -- control flow obfuscation and import obfuscation -- provide additional layers of protection.

Key Capabilities

- SHA-256 stream cipher string encryption with per-file random keys and HMAC integrity
- Bytes literal encryption (binary data, keys, tokens)
- f-string constant encryption (constant parts within f-strings)
- HMAC-SHA256 anti-tamper verification on encrypted string/bytes tables
- Aggressive name mangling (ALL user-defined identifiers by default)
- Control flow obfuscation with opaque predicates and dead code injection (optional)
- Import obfuscation via encrypted __import__ calls (optional)
- Docstring and comment stripping via AST transformation
- Type annotation removal (return types, argument types, variable annotations)
- Native compilation via Nuitka integration (optional Phase 4)
- Binary plaintext leak scanning (optional Phase 5)
- Already-obfuscated detection via PIPpro signature comment (with legacy _K fallback)

- Configurable preserve lists for public API names
- Deterministic output via PRNG seed for reproducible builds
- Strict mode (--strict) for CI/CD pipelines, default when compiling
- Secure temp staging (mkdtemp + atexit cleanup) for compilation
- Zero external dependencies for the obfuscation pipeline (stdlib only)

Design Philosophy

PIPpro applies defense in depth to source code protection. Each phase addresses a different attack surface: stripping removes human-readable documentation, encryption removes string-based intelligence with cryptographic-strength SHA-256 stream cipher and HMAC integrity, control flow obfuscation defeats static analysis, import obfuscation hides dependency graphs, renaming removes semantic meaning from ALL identifiers, compilation removes Python source entirely, and verification confirms the binary is clean. The phases compose naturally -- each one makes the next more effective.

2. System Requirements

Obfuscation (Phases 1-3.5)

- Python 3.10 or later
- No additional packages required (stdlib only: ast, os, secrets, hashlib, hmac, argparse)
- Works on Windows, Linux, and macOS

Compilation (Phase 4)

- Nuitka (pip install nuitka)
- C compiler: MinGW64 (Windows), GCC (Linux), Clang (macOS)
- Minimum 4 GB RAM for standalone builds (8 GB recommended)

Verification (Phase 5)

- Compiled binary from Phase 4
- No additional dependencies

3. Installation

Windows -- Installer (.exe)

Download PIPpro_Setup_2.0.0.exe from densedefense.com. Run the installer with administrator privileges. The installer:

- Installs pippro.exe to C:\Program Files\PIPpro\
- Adds the install directory to the system PATH
- Creates a registry key at HKLM\SOFTWARE\PIPpro with InstallPath and Version
- Uses LZMA2/Ultra64 compression (solid archive)

After installation, open a new terminal and run:

```
pippro --help
```

Linux -- Debian Package (.deb)

Download pippro_2.0.0_amd64.deb from densedefense.com. Install:

```
sudo dpkg -i pippro_2.0.0_amd64.deb
```

The .deb package installs the PIPpro source to /opt/pippro/pippro.py and creates a launcher script at /usr/local/bin/pippro. Requires Python 3.10 or later. Uninstall with:

```
sudo dpkg -r pippro
sudo dpkg -P pippro   # also removes /opt/pippro
```

Manual / Portable

PIPpro is a single Python file with zero dependencies. Copy obfuscate_source.py anywhere and run directly:

```
python obfuscate_source.py --help
```

4. Quick Start

Obfuscate a directory of Python files to a new output directory:

```
pippro --source-dir src/ --output-dir dist/ --encrypt-strings
```

Obfuscate specific files in-place (back up first!):

```
pippro --files app.py lib.py --in-place --encrypt-strings
```

Preview what would happen without writing files:

```
pippro --source-dir src/ --dry-run --show-map --show-string-stats
```

Full pipeline: obfuscate, compile to native binary, verify no leaks:

```
pippro --files app.py \
  --compile --standalone --entry app.py \
  --verify --verify-strings "password" "api_key"
```

Maximum protection: all phases enabled:

```
pippro --files app.py \
  --rename-all --control-flow --obfuscate-imports \
  --compile --strict --standalone --entry app.py \
  --verify --verify-strings "password" "api_key"
```

Conservative mode (v1.x rename behavior, underscore-only):

```
pippro --files app.py --in-place --safe-rename-only --encrypt-strings
```

5. CLI Reference

Input Selection (mutually exclusive, one required)

Flag	Description
--source-dir DIR	Process all .py files in DIR recursively (skips . and __ prefixed dirs)
--files F1 F2 ...	Process specific .py files

Output Mode (at least one required)

Flag	Description
--output-dir DIR	Write obfuscated files to DIR (preserving subdirectory structure)
--in-place	Overwrite source files in-place (BACK UP FIRST)
--dry-run	Analyze without writing any files
--compile	Compile obfuscated source to native binary (implies staging)

String Encryption Options

Flag	Description
--encrypt-strings	Enable SHA-256 stream cipher string encryption (default: on)
--no-encrypt-strings	Disable string encryption
--min-string-length N	Minimum string length to encrypt (default: 3)

Name Mangling Options (NEW in v2.0.0)

Flag	Description
--rename-all	Rename ALL user-defined names (default: on in v2.0.0)
--safe-rename-only	Conservative mode: only rename _-prefixed names (v1.x behavior)

Control Flow Obfuscation (NEW in v2.0.0)

Flag	Description
--control-flow	Enable opaque predicates, dead code injection, closure wrapping

Import Obfuscation (NEW in v2.0.0)

Flag	Description
--obfuscate-imports	Replace non-stdlib imports with encrypted __import__ calls

Strict Mode (NEW in v2.0.0)

Flag	Description
--strict	Abort on any file processing error (default when --compile is used)

Name Preservation

Flag	Description
--preserve NAMES	Comma-separated list of names to never rename
--preserve-file FILE	File with names to never rename (one per line, # comments)

Diagnostics

Flag	Description
--seed N	PRNG seed for deterministic, reproducible rename maps
--show-map	Print the complete original -> mangled name mapping
--show-string-stats	Print string encryption statistics (encrypted/skipped counts)

Compilation (Phase 4)

Flag	Description
--compile	Invoke Nuitka after obfuscation
--standalone	Bundle Python interpreter + deps into standalone binary (default)
--module	Compile as importable .so (Linux) / .pyd (Windows) module
--entry FILE	Main entry point for compilation (default: first file)
--compile-output DIR	Nuitka output directory (default: dist_pippro)
--include-data SRC=DEST	Data directories/files to bundle (repeatable)
--include-module MOD	Modules to explicitly include in compilation
--exclude-module MOD	Modules to exclude from compilation
--nuitka-args ARG ...	Extra arguments passed directly to Nuitka

Verification (Phase 5)

Flag	Description
--verify	Scan compiled binary for plaintext leaks (requires --compile)
--verify-strings S ...	Additional sensitive strings to check for in the binary

6. Obfuscation Features Overview

PIPpro v2.0.0 applies seven protection phases in pipeline order. Each phase transforms the code further, and the phases are designed to compose: encryption happens before renaming so that the decryptor function names themselves get mangled, making them indistinguishable from application code. Two new optional phases (2.5 and 3.5) add control flow and import obfuscation.

Phase	Name	What It Does	Dependencies
1	Strip	Remove docstrings, type annotations, comments	None (stdlib ast)
2	Encrypt	SHA-256 stream cipher string/bytes encryption + HMAC	None (stdlib hashlib, hmac)
2.5	Control Flow	Opaque predicates, dead code injection (optional)	None (stdlib ast)
3	Rename	Mangle ALL user-defined identifiers (aggressive)	None (stdlib ast)
3.5	Import Obfusc.	Encrypt non-stdlib import statements (optional)	None (stdlib ast)
4	Compile	Compile to native binary via Nuitka	Nuitka + C compiler
5	Verify	Scan binary for plaintext string leaks	None

Phases 1-3.5 are always available (stdlib only). Phases 4-5 are optional and require Nuitka. All seven phases can be invoked in a single CLI command.

7. String Encryption (Phase 2) -- SHA-256 Stream Cipher

PIPpro v2.0.0 replaces every string and bytes literal in the source with an encrypted table lookup using a SHA-256 key-derived stream cipher. At build time, a 32-byte master key is generated per file from os.urandom(32). Every string longer than the minimum length (default 3 characters) is encrypted using a SHA-256 counter-mode stream derived from the master key and the string's table index. The original string literal is replaced with a function call: _s(index) for strings, _b(index) for bytes.

> **Upgrade from v1.x**
> PIPpro v1.x used simple XOR encryption (key byte cycling). v2.0.0 uses SHA-256 in counter mode: stream_block[n] = SHA256(master_key || index || n). This eliminates repeating-key weaknesses and provides a cryptographically derived keystream. The upgrade was a P0 HallMonitor recommendation.

SHA-256 Stream Cipher

For each string at table index i, PIPpro derives a unique keystream by computing SHA-256(K || i || block_num) for sequential block numbers. Each SHA-256 hash produces 32 bytes of keystream. The plaintext is XORed with this derived stream. Because each string gets a unique (index, block) pair, no two strings share keystream material, even within the same file.

Runtime Decryptors

PIPpro injects a preamble at the top of each processed module containing:

- _PP: PIPpro signature marker (for already-obfuscated detection)
- import hashlib, hmac (runtime dependencies for decryption)
- _K: The 32-byte master key (bytes literal)
- _S: List of encrypted string ciphertexts
- _H: HMAC-SHA256 digest over the concatenated string table (anti-tamper)
- _C: Cache dictionary for decrypted strings (decrypt once, reuse)
- _V: Verification flag (HMAC checked once on first decryption)
- _s(i): Decryptor function -- verifies HMAC, derives SHA-256 stream, decrypts _S[i], caches in _C
- _B: List of encrypted bytes ciphertexts (if bytes literals present)
- _HB: HMAC-SHA256 digest over the bytes table
- _D: Cache dictionary for decrypted bytes
- _VB: Verification flag for bytes table
- _b(i): Decryptor function for bytes literals (with separate HMAC verification)

The decryptor cache ensures each string is decrypted exactly once at runtime. Subsequent accesses return the cached plaintext from the dictionary, so there is negligible performance overhead after first use.

f-string Encryption

PIPpro also encrypts constant parts within f-strings. An f-string like f"hello {name} world" has constant parts ("hello " and " world") and expression parts ({name}). PIPpro encrypts the constant parts by converting them to FormattedValue nodes calling _s(), resulting in f"{_s(0)}{name}{_s(1)}". This prevents string extraction from recovering partial f-string fragments.

Deduplication

If the same string appears multiple times in a file, it is stored once in the encrypted table and all references point to the same index. This keeps the encrypted table compact and the decryptor cache effective.

8. HMAC-SHA256 Integrity Checking (NEW in v2.0.0)

PIPpro v2.0.0 computes an HMAC-SHA256 digest over each encrypted table (strings and bytes separately). The HMAC is stored alongside the table and verified at runtime before the first decryption attempt.

How It Works

- At build time: all encrypted entries are concatenated and HMAC-SHA256(K, concat) is computed
- The HMAC digest is stored as _H (string table) and _HB (bytes table)
- At runtime: the first call to _s() or _b() triggers HMAC verification
- If verification fails: RuntimeError('integrity') is raised immediately
- After verification: a flag (_V / _VB) is set to skip re-verification

This prevents an attacker from modifying the encrypted table entries (e.g., zeroing them out to reveal patterns) without detection. Any tampering with the encrypted data, the HMAC digest, or the master key will cause the integrity check to fail at startup.

Anti-Tamper Protection
The HMAC-SHA256 integrity check was a P1 HallMonitor recommendation. Without it, an attacker could modify encrypted entries and observe behavior changes to infer string content. With HMAC, any modification crashes the application immediately.

9. Name Mangling (Phase 3) -- Aggressive by Default

PIPpro v2.0.0 renames ALL user-defined identifiers to random 12-character strings like _a7kx9m2p4b1q. This removes all semantic meaning from function names, class names, variables, and parameters.

Breaking Change from v1.x
PIPpro v1.x only renamed underscore-prefixed names (_helper, _parse). v2.0.0 renames ALL user-defined names by default (--rename-all is ON). Use --safe-rename-only to revert to v1.x conservative behavior. This was a P0 HallMonitor recommendation.

What Gets Renamed (--rename-all, default)

- ALL functions and methods defined in the source
- ALL classes defined in the source
- ALL variables assigned in the source
- ALL function arguments
- Global/nonlocal declarations referencing renamed identifiers
- Intra-project import aliases (from module import func)

What Gets Renamed (--safe-rename-only, v1.x behavior)

- Functions and methods with underscore-prefixed names (_helper, _parse, etc.)
- Classes with underscore-prefixed names (_InternalParser)
- Variables with underscore-prefixed names (_count, _buffer)
- Function arguments that are underscore-prefixed

What Is Never Renamed

- Python builtins (print, len, dict, list, etc.)
- Python keywords and soft keywords (if, for, match, case, etc.)
- Standard library module names (os, sys, ast, etc.)
- Dunder names (__init__, __version__, __name__, etc.)
- Special parameters (self, cls, args, kwargs)
- Names in the preserve list (--preserve or --preserve-file)
- Names not defined in the source (external library names)
- Attribute accesses (obj.method) -- only bare Name nodes are renamed

Cross-File Consistency

When processing multiple files (--source-dir or multiple --files), PIPpro collects all defined names across all files before generating the rename map. This ensures that if module A defines helper() and module B imports it, both get the same mangled name. The rename map is global across the entire processing batch.

10. Control Flow Obfuscation (Phase 2.5) -- NEW in v2.0.0

PIPpro v2.0.0 introduces optional control flow obfuscation via the --control-flow flag. This phase transforms the program's control flow graph to defeat static analysis and decompilers.

Opaque Predicates

Every if/else condition is wrapped in a lambda closure that returns a single-element list. The original condition if X: becomes if (lambda: [X])()[0]: -- semantically identical but opaque to static analyzers that cannot evaluate the closure.

Dead Code Injection

PIPpro inserts unreachable code branches using mathematical tautologies that are always False. For example: if (7 * 13) % 91 != 0: _d1 = None. Since 7*13=91 and 91%91=0, the condition is always False, but a static analyzer must prove this to eliminate the branch.

- Dead code is injected before ~30% of function body statements
- Up to 3 dead code branches are added at module level (after every 5th statement)
- 5 different tautology patterns are rotated to avoid signature detection

Impact on Analysis

Control flow obfuscation increases the cyclomatic complexity of the code and forces decompilers to reason about lambda closures and unreachable branches. Combined with name mangling and string encryption, this makes automated code analysis prohibitively expensive.

11. Import Obfuscation (Phase 3.5) -- NEW in v2.0.0

PIPpro v2.0.0 introduces optional import obfuscation via the --obfuscate-imports flag. This phase replaces standard import statements with encrypted __import__ calls, hiding the dependency graph.

How It Works

- import foo becomes _mN = __import__(_s(idx)) where idx points to encrypted 'foo'
- from foo import bar becomes bar = getattr(__import__(_s(idx1)), _s(idx2))
- Module names are added to the encrypted string table and decrypted at runtime via _s()
- Standard library imports are preserved unchanged (os, sys, hashlib, etc.)
- Star imports (from foo import *) are preserved unchanged for safety
- Multi-name imports (import a, b) are preserved unchanged for safety

Security Benefit

Without import obfuscation, the import statements at the top of each file reveal the application's dependency graph -- which libraries it uses, which internal modules exist, and how they connect. Import obfuscation removes this intelligence. An attacker examining the obfuscated source sees only __import__(_s(42)) with no indication of what module is being loaded.

12. Docstring & Annotation Stripping (Phase 1)

The first pipeline phase removes all human-readable documentation from the AST before any other transformation occurs.

Docstrings

Module-level, class-level, and function-level docstrings are removed. If removing a docstring leaves an empty body (e.g., a class with only a docstring), a pass statement is inserted to maintain valid syntax.

Type Annotations

- Function return type annotations (def f() -> int:) are removed
- Argument type annotations (def f(x: int):) are removed
- *args and **kwargs annotations are removed
- Variable annotations with values (x: int = 5) become plain assignments (x = 5)
- Variable annotations without values (x: int) are removed entirely

13. Native Compilation (Phase 4)

PIPpro optionally invokes Nuitka to compile obfuscated Python source into a native binary. This eliminates Python source code entirely -- the binary contains only compiled C code.

Modes

- **Standalone (--standalone):** Bundles the Python interpreter and all dependencies into a self-contained directory. The output is a single directory with the executable and all required shared libraries.
- **Module (--module):** Compiles a Python file as an importable .so (Linux) or .pyd (Windows) shared library. Useful for protecting individual modules within a larger Python application.

Data Inclusion

Use --include-data to bundle non-Python files (templates, config files, static assets) into the compiled distribution:

```
pippro --files app.py --compile --include-data templates=templates
```

Secure Temp Staging (NEW in v2.0.0)

When --compile is used without --output-dir or --in-place, PIPpro automatically creates a secure temporary staging directory using tempfile.mkdtemp(prefix='pippro_'). An atexit handler ensures the staging directory is cleaned up on exit (including abnormal termination). This prevents obfuscated source from lingering in temporary locations.

14. Binary Verification (Phase 5)

After compilation, PIPpro can scan the resulting binary for plaintext string leaks. This catches cases where strings survived the encryption pipeline (e.g., strings below the minimum length, or strings introduced by Nuitka itself).

How It Works

- Reads the binary file into memory
- Extracts all printable ASCII sequences of 6+ characters (like the Unix strings command)
- Checks extracted strings against built-in sensitive patterns
- Checks against user-specified sensitive strings (--verify-strings)
- Reports total string count, leak count, and leak details

Built-in Sensitive Patterns

Pattern Category	Examples Detected
System commands	sudo, chmod, chown, iptables, ufw, systemctl, sshd, passwd, useradd
Remediation keys	automation_command, rollback_command
Credential patterns	password=, passwd:, secret=, token=, api.key=
Internal paths	/opt/fortefide/, /opt/fortefed/

Leaks are deduplicated by content. A clean scan reports CLEAN; a failed scan lists each unique leak with its category and the first 80 characters of the matching string.

15. Preserve Lists

With v2.0.0's aggressive renaming (--rename-all), preserve lists are more important than ever. Any name that is part of a public API, referenced by external code, or used in cross-module imports must be preserved. PIPpro supports two ways to specify preserved names.

Inline (--preserve)

```
pippro --files app.py --in-place --preserve "Scanner,ScanResult,run_scan"
```

File (--preserve-file)

Create a text file with one name per line. Lines starting with # are comments:

```
# preserve_names.txt
# Flask / app.py
app
socketio

# Scanner public API
Scanner
ScanResult
scan_host
run_scan
```

```
pippro --source-dir src/ --output-dir dist/ --preserve-file preserve_names.txt
```

Automatic Safe List

PIPpro automatically protects the following categories of names regardless of preserve lists:

- All Python builtins (dir(builtins))
- All Python keywords and soft keywords
- All standard library module names (sys.stdlib_module_names)
- self, cls, args, kwargs
- All dunder names (__init__, __version__, etc.)
- Names not defined anywhere in the source files being processed

16. Strict Mode & Secure Staging (NEW in v2.0.0)

Strict Mode (--strict)

When --strict is enabled, PIPpro aborts immediately on any file processing error instead of collecting errors and continuing. This is the default behavior when --compile is used, ensuring that compilation never proceeds with partially-obfuscated source.

```
# Explicit strict mode for CI/CD
pippro --source-dir src/ --output-dir dist/ --strict

# Implicit strict mode (--compile implies --strict)
pippro --files app.py --compile --entry app.py
```

Rename Map Protection Warning

When --show-map is used with --in-place, PIPpro prints a warning to stderr: the rename map printed to stdout could be captured in CI/CD logs, exposing the mapping between original and mangled names. Use --output-dir instead of --in-place to avoid this risk.

17. Dry Run & Diagnostics

Use --dry-run to analyze your source without writing any files. Combine with --show-map and --show-string-stats for full visibility.

```
pippro --source-dir src/ --dry-run --show-map --show-string-stats

========================================
 PIPpro v2.0.0
 Python IP Protection
 HallMonitor-hardened
========================================

  Files: 7
    src/app.py
    src/scanner.py
    ...

  String encryption: ON (SHA-256 stream cipher + HMAC)
  Rename mode: AGGRESSIVE (all user-defined names)
  Control flow: OFF
  Import obfuscation: OFF
  Min string length: 3

  Names mangled:      187
  Strings encrypted:  387
  Bytes encrypted:    12
  Files processed:    7

  Rename map:
    buffer                   -> _a7kx9m2p4b1q
    execute_command          -> _c2dn8f4r1t7w
    Scanner                  -> _p8rv3j6x2k0m
    ...

  String stats:
    Encrypted: 387 strings, 12 bytes literals
    Skipped:   31 (below min length)

  (dry run -- no files written)
```

18. Integration with Build Pipelines

PIPpro is designed to slot into existing build workflows. The typical pattern is: back up source, obfuscate in-place, build with Nuitka, restore originals.

Windows Batch Script Pattern

```
@echo off
REM 1. Back up originals
mkdir _backup_src
copy app.py _backup_src\
copy scanner.py _backup_src\

REM 2. Obfuscate in-place (v2.0.0: aggressive rename + control flow)
pippro --files app.py scanner.py --in-place \
  --encrypt-strings --rename-all --control-flow \
  --preserve-file preserve_names.txt

REM 3. Build with Nuitka
python -m nuitka --standalone app.py
```

```
REM 4. Restore originals
copy /y _backup_src\app.py .
copy /y _backup_src\scanner.py .
rmdir /s /q _backup_src
```

Single-Command Pipeline (Recommended)

Use PIPpro's built-in --compile with secure temp staging:

```
pippro --source-dir src/ \
  --compile --standalone --entry src/app.py \
  --rename-all --control-flow --obfuscate-imports \
  --include-data templates=templates \
  --include-module scanner \
  --verify --verify-strings "password" "api_key"
```

CI/CD Integration

PIPpro's --seed flag enables deterministic builds for CI/CD pipelines. Given the same seed, the same source files will always produce the same rename map. --strict mode (default with --compile) ensures any error fails the build:

```
pippro --source-dir src/ --output-dir dist/ --seed 42 \
  --encrypt-strings --strict
```

19. Worked Examples

Example 1: Protect a Flask Application

```
# Obfuscate everything, preserve Flask route handlers
pippro --source-dir myapp/ --output-dir protected/ \
  --encrypt-strings --rename-all \
  --preserve "app,create_app,db,login_required"
```

Example 2: Maximum Protection

```
# All phases: encrypt + control flow + import obfusc + compile + verify
pippro --files algorithms/pricing.py \
  --rename-all --control-flow --obfuscate-imports \
  --compile --standalone --entry algorithms/pricing.py \
  --verify
```

Example 3: Full Security Audit

```
# Obfuscate, compile, then verify no secrets leaked
pippro --source-dir src/ \
  --compile --standalone --entry src/main.py \
  --verify --verify-strings "SECRET_KEY" "DB_PASSWORD" "AWS_ACCESS"
```

Example 4: ForteFide Build (Real-World)

DenseDefense uses PIPpro to protect ForteFide and ForteFed before every release build. The actual production command:

```
python obfuscate_source.py \
  --source-dir fortefide_remediation \
  --in-place --encrypt-strings \
  --rename-all --control-flow \
  --preserve-file build_tools/preserve_names.txt
```

This encrypts 8,800+ strings across the remediation module and scanner files with SHA-256 stream cipher + HMAC integrity. Aggressive renaming covers all identifiers. The preserve list ensures Flask routes, public API names, and cross-module imports remain intact.

Example 5: Conservative Mode (v1.x Behavior)

```
# Only rename _-prefixed names, no control flow or import obfuscation
pippro --source-dir src/ --output-dir dist/ \
  --safe-rename-only --encrypt-strings
```

20. HallMonitor Security Scorecard

The HallMonitor security advisor performed a comprehensive audit of PIPpro v1.2.0 and identified 8 findings across P0 (critical), P1 (high), and P2 (medium) severity levels. PIPpro v2.0.0 addresses ALL findings.

Score Improvement

Metric	v1.2.0	v2.0.0
Overall Score	4.4 / 10	7.4 / 10
String Encryption	XOR (weak)	SHA-256 stream cipher (strong)
Integrity Checking	None	HMAC-SHA256 anti-tamper
Name Mangling	_-prefixed only	ALL names (--rename-all default)
Control Flow	Not available	Opaque predicates + dead code
Import Protection	Not available	Encrypted __import__ calls
Temp File Security	No cleanup	mkdtemp + atexit cleanup
CI/CD Safety	No strict mode	--strict default for --compile
Rename Map Exposure	No warning	stderr warning with --in-place

Findings Addressed

ID	Priority	Finding	Status
HM-01	P0	XOR encryption trivially reversible	FIXED: SHA-256 stream cipher
HM-02	P0	Only _-prefixed names renamed	FIXED: --rename-all default
HM-03	P1	No integrity check on encrypted tables	FIXED: HMAC-SHA256
HM-04	P1	No control flow obfuscation	FIXED: --control-flow phase
HM-05	P1	Temp staging dir not cleaned up	FIXED: mkdtemp + atexit
HM-06	P2	Import statements reveal dependencies	FIXED: --obfuscate-imports
HM-07	P2	No strict mode for CI/CD	FIXED: --strict flag
HM-08	P2	Rename map printable to stdout logs	FIXED: stderr warning

21. Troubleshooting

Already-Obfuscated Detection

PIPpro v2.0.0 detects already-obfuscated files using the PIPpro signature comment (_PP = '# PIPpro-obfuscated'). For backward compatibility, it also detects the legacy v1.x XOR key preamble (_K = b'...'). If detected, the file is skipped with a SKIP message. This prevents double-encryption which can cause Nuitka memory explosion during compilation.

SyntaxError After Obfuscation

PIPpro verifies every obfuscated file by parsing it with ast.parse() before writing. If you see a SyntaxError, it will be reported in the errors list at the end of the run. Common causes:

- Source file has existing syntax errors
- Python version mismatch (file uses 3.12 features, PIPpro runs on 3.10)

ImportError After Obfuscation

If code fails at runtime after obfuscation, the most common cause is a missing preserve entry. With --rename-all (v2.0.0 default), ALL user-defined names are renamed. If module A exports my_func and module B imports it by name, both must be processed together (--source-dir) or my_func must be in the preserve list.

HMAC Integrity Error at Runtime

If you see RuntimeError('integrity') at application startup, the encrypted string table has been modified after obfuscation. This can happen if the obfuscated file is edited manually, or if a post-processor modifies bytes literals. Re-obfuscate from clean source to fix.

Nuitka Not Found

Phase 4 requires Nuitka. Install with: pip install nuitka. PIPpro checks for nuitka in PATH first, then tries python -m nuitka.

22. Licensing & Support

PIPpro is a licensed product from DenseDefense. The obfuscation engine (Phases 1-3.5) has zero external dependencies and requires only a valid PIPpro license. Phase 4 (compilation) uses Nuitka, which has its own licensing terms.

Product: PIPpro -- Python IP Protection

Version: 2.0.0 (HallMonitor-hardened)

Publisher: DenseDefense -- DFW, Texas
Website: densedefense.com
Support: support@densedefense.com
License: Commercial -- per-developer or site license