Proof of Concept (PoC): pycdas & pycdc

Intern Name: Akkshat Shah
Internship ID: 261
Tool ID: 248, 249
Date: July 2025

🛠️ Tool Name

pycdas & pycdc

📜 History

The pycdas and pycdc tools emerged from the need to analyze and reverse-engineer Python bytecode, particularly when original source code is not available. With the widespread adoption of Python in automation, scripting, and malicious tools, these utilities became essential for digital forensics and malware analysis, allowing investigators to retrieve source logic from compiled Python files (.pyc/.pyo). Maintained as open-source, the tools are supported by a community of reverse engineers and forensic practitioners.

📄 Description

pycdc is a decompiler focused on transforming Python compiled bytecode files (.pyc, .pyo) back into readable high-level Python source.
pycdas operates as a disassembler, producing opcode-level human-readable representations of Python bytecode. Together, they provide in-depth insight into the behavior and structure of compiled Python executables, supporting both static analysis and recovery tasks.

📌 What Is This Tool About?

pycdas/pycdc address the gap between compiled Python artifacts and human-level understanding, particularly where source code is lost, hidden, or aggressively obfuscated. These tools allow practitioners to:

- Retrieve code from compiled files for analysis, auditing, or legal review.

- Dissect and examine possibly malicious or suspicious Python-based executables.
- Reconstruct logic flow and variable usage from obfuscated, packed, or custom-distributed Python applications.

⭐ Key Characteristics / Features

- Accurate decompilation of .pyc/.pyo files to readable Python source
- Opcode-level bytecode disassembly for deep inspection
- Compatibility with Python 2.x and 3.x file formats
- Batch processing support for analyzing directories of files
- CLI and scriptable workflows for automation
- Annotated output highlighting control flow, imports, constants, and string usage
- Identification of code objects, classes, and functions
- Handles bytecode generated from multiple operating systems
- Minimal dependencies; fast, portable, and lightweight
- Open-source license supporting custom integration
- Optional annotated outputs for educational or audit purposes

🔧 Types / Modules Available

- pycdc Decompiler Core: Decompiles bytecode to source code.
- pycdas Disassembler: Converts bytecode to opcode-level annotated listings.
- Batch Analyzer: Automates mass decompilation/disassembly tasks.
- Opcode Visualization Module: Displays code control and data flow.
- File Integrity & Packing Detector: Flags signs of obfuscation or modification.

🎯 How Will This Tool Help?

- Recover Lost Code: Bring back source from compiled distributions during forensics or IP audits.
- Malware Analysis: Safely analyze obfuscated or suspicious .pyc payloads in malware cases.
- Compliance Verification: Check distributed code compliance when only .pyc files are available.
- Incident Response: Reconstruct deleted, modified, or malicious automation logic.
- Software Auditing: Review legacy, abandoned, or third-party “code only” deployments.

📸 Proof of Concept (PoC) Images

- Decompilation of a .pyc file to readable Python code

```
/0x17.c# uncompyle6
No input files given to decompile

Usage:
  uncompyle6 [OPTIONS]... [ FILE | DIR]...
  uncompyle6 [--help | -h | --V | --version]

Examples:
  uncompyle6      foo.pyc bar.pyc       # decompile foo.pyc, bar.pyc to stdout
  uncompyle6 -o . foo.pyc bar.pyc       # decompile to ./foo.pyc_dis and ./bar.pyc_dis
  uncompyle6 -o /tmp /usr/lib/python1.5 # decompile whole library

Options:
  -o <path>     output decompiled files to this path:
                if multiple input files are decompiled, the common prefix
                is stripped from these names and the remainder appended to
                <path>
                  uncompyle6 -o /tmp bla/fasel.pyc bla/foo.pyc
                    -> /tmp/fasel.pyc_dis, /tmp/foo.pyc_dis
                  uncompyle6 -o /tmp bla/fasel.pyc bar/foo.pyc
                    -> /tmp/bla/fasel.pyc_dis, /tmp/bar/foo.pyc_dis
                  uncompyle6 -o /tmp /usr/lib/python1.5
                    -> /tmp/smtplib.pyc_dis ... /tmp/lib-tk/FixTk.pyc_dis
  --compile | -c <python-file>
                attempts a decompilation after compiling <python-file>
  -d            print timestamps
  -p <integer>  use <integer> number of processes
  -r            recurse directories looking for .pyc and .pyo files
  --fragments   use fragments deparser
  --verify      compare generated source with input byte-code
  --verify-run  compile generated source, run it and check exit code
  --syntax-verify compile generated source
  --linemaps    generated line number correspondencies between byte-code
                and generated source output
  --encoding  <encoding>
                use <encoding> in generated source according to pep-0263
  --help        show this message

Debugging Options:
```

- Disassembly output showing annotated Python opcodes

```
PS C:\druva_extracted> uncompyle6.exe -o decompiled -r .\converted\inSyncClient\
.\converted\inSyncClient\ctypeExtension.pyc --
.\converted\inSyncClient\states\__init__.pyc -- decompiled 76 files: 76 okay, 0 failed
# decompiled 76 files: 76 okay, 0 failed
```

🗒️ 15-Liner Summary

1. Extracts readable source from Python bytecode (.pyc/.pyo)
2. Provides opcode-level disassembly
3. Supports major Python 2.x and 3.x versions
4. Batch file and directory processing
5. Detects and annotates classes, functions, imports
6. Highlights strings and constants for review
7. Portable and requires minimal setup
8. Fast execution, low overhead
9. CLI-driven; easy scripting for bulk operations
10. Open-source, cross-platform
11. Useful for malware, legal, and audit investigations
12. Integrates into forensics suites or custom pipelines
13. Produces annotated outputs for teaching or documentation
14. Handles partially corrupted or packed bytecode files
15. Widely used by digital forensics and reverse engineering specialists

⏱️ Time to Use / Best Case Scenarios

- When only compiled Python executables are available
- During digital forensics involving suspect Python scripts
- Code auditing for compliance/IP review of .pyc-based distributions
- Malware investigations containing Python payloads
- Automated analysis of code dumps from compromised infrastructure

🕵️ When to Use During Investigation

- Incident Response: Rapid review of automation or dropped .pyc files.
- Malware Analysis: Unpacking and understanding Python-based malware.
- Digital Forensics: Recovering deleted source code or reconstructing attacker logic.
- Legal/IP Review: Verifying compliance or investigating infringement.
- Audit: Reviewing third-party or legacy systems shipped only as .pyc.

👨‍💻 Best Person to Use This Tool & Required Skills

Ideal Users:
- Digital Forensics Examiners
- Malware Analysts
- Incident Responders
- Reverse Engineers

Required Skills:
- Familiarity with Python programming paradigms
- Understanding of Python bytecode structure
- Static analysis and basic reverse engineering principles
- Proficiency with command-line tools and script automation

🧩 Flaws & Suggestions to Improve

- May struggle with heavily obfuscated/protected .pyc files
- Limited support for brand-new/experimental Python features
- No native GUI; less accessible for non-technical users
- Minimal real-time monitoring or cloud .pyc analysis
- Visualization (control/data flow) remains basic
- Plug-in system for extending to niche/custom .pyc variants would improve flexibility

✅ Good About the Tool

- Lightweight, open-source, and portable
- Cross-platform; works on all major operating systems
- High decompilation accuracy for standard .pyc files
- Supports both one-off and bulk/batch operations
- Seamless CLI integration for advanced automation
- Essential in reverse engineering, auditing, and forensic scenarios

pycdas and pycdc offer crucial capabilities for Python bytecode inspection, making them essential in digital forensics, security analysis, and source recovery across platforms and investigations.