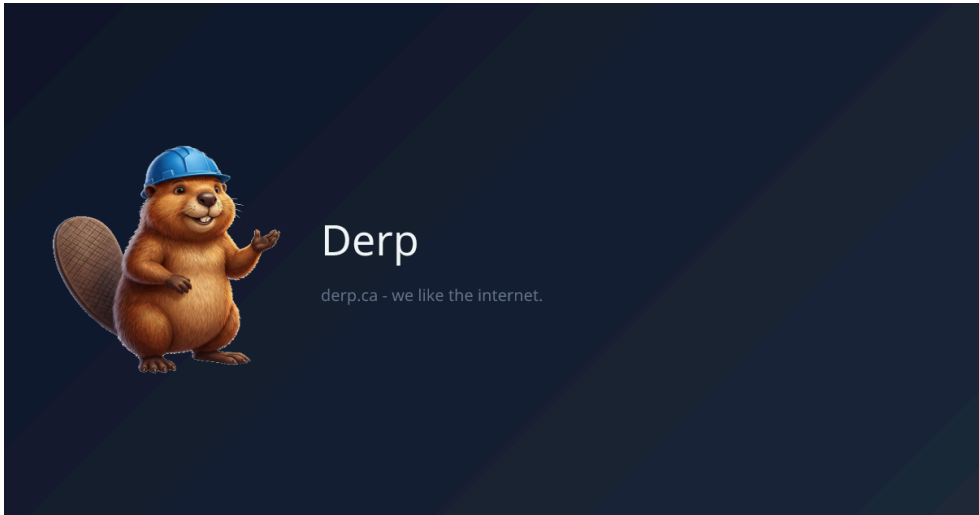


Python Loader Evolution: Five Encryption Generations



malware python donut incident-response purecoder obfuscation

Every RAT in this campaign -- nine families across six parallel attack chains -- entered through the same pipeline. A batch file downloads a ZIP from a disposable Cloudflare tunnel. The ZIP contains a Python runtime and one or more loader scripts. Each loader decrypts embedded shellcode, and that shellcode bootstraps the .NET Common Language Runtime (CLR) to load the actual payload. The RATs themselves barely changed over 176 days. The delivery pipeline evolved five times.

This post is the fifth and final entry in the [SERPENTINE#CLOUD](#) (tracked by Securonix, June 2025) breach analysis series. Prior posts covered the inner layers: [PureLogs](#) (the infostealer), [PureCrypter](#) (the .NET 3DES loader), [Violet RAT](#) (the 120-command dispatcher), and [Remcos](#) (Autolt persistence and banking fraud). This one covers the outer layers -- batch download, Python loader, Donut shellcode, then .NET handoff.

Intelligence refresh (2026-02-26)

We re-validated the Dec17 and Feb1 stager URL mapping from decoded batch scripts and refreshed sandbox coverage for representative Python loaders from the Nov10, Nov19, and Nov24 sets.

Artifact	SHA-256	VT	Tria.ge
BKSNOLazyNov10_Viooooooooo.py	c9dc14435857fd00e3f81e219c362c4dc235774dea08fb49fca874dfa108cdd2	Not indexed	https://tria.gsyqahsb15c
1vio-obf.py (EMP Kramer)	97e71a0347439a169a3278ca85c11f28fe2098c42f7513e2621c0a3f3258b7f2	Not indexed	https://tria.gsyq7tab15f
1uunov24.py	ced8796ae647584824c9907d0513f2d2f4b63f9b5bb9cf713e5a59a777335824	1/76	https://tria.gsyrcab15g

The delivery stack

```
Batch stager (Cloudflare tunnel download)
-> Python loader (5 encryption generations)
-> Donut shellcode (Chaskey CTR, AMSI bypass)
    -> PureCrypter (.NET 3DES loader) -- covered in post 2
    -> Inner RAT -- covered in posts 1, 3, 4
```

Seven waves

Wave	Date	Encryption	Execution	Obfuscation	Donut	Loaders
1	Oct 4-5	Double-XOR (16B keys) + RC4 (8-char)	APC into notepad.exe / in-process CFUNCTYPE	Junk variables, readable	v0.9.2 (x86+x64)	4
2	Nov 10	AES-256-CBC + dual XOR (16B keys)	Early Bird APC into explorer.exe	None (plaintext .py)	v0.9.2 (x64)	7
3	Nov 10	AES-256-CBC + dual XOR (different keys)	Early Bird APC into explorer.exe	None (plaintext .py)	v0.9.2 (x64)	7
4	Nov 19	RC4 (8-char key), Kramer .pyc	In-process CFUNCTYPE	Kramer bytecode obfuscator	v0.9.3 (x64)	5
5	Nov 19-20	RC4 (8-char key), Kramer .pyc	In-process CFUNCTYPE	Kramer bytecode obfuscator	v0.9.3 (x64)	10
6	Dec 17	Same as Wave 1 (redeployed)	Same as Wave 1	Same as Wave 1	v0.9.2	2
7	Dec 16	Mixed AES+XOR and RC4	Mixed APC and in-process	Mixed plaintext and Kramer	Mixed	13

Seven waves, 48 loader files, the same five RAT families repackaged over and over. Each wave either upgrades the encryption, changes the injection method, or both.

Wave 1 (October 5) -- XOR + RC4

The first deployment used two distinct loader architectures. All four files deliver the same .NET payload (`Fviwknzr.exe`, the PureCrypter loader analyzed in [post 2](#)). Python version: **3.14** (pre-release at time of deployment) -- the only wave to use it. All later waves drop to 3.12.

Type A: MOF loaders (double-XOR + process injection)

Two files (`Oct05_MOF.py`, `Start_Oct05_MOF.py`), each ~841 KB. Shellcode is XOR-encrypted with two 16-byte keys, then base64-encoded:

```
base64_decode -> XOR(key2) -> XOR(key1) -> raw shellcode
```

File	XOR Key 1	XOR Key 2
<code>Oct05_MOF.py</code>	<code>896a01e960d97a791133c51c7bf6629a</code>	<code>2faa926e690a4fd4e7c5c1acbb910a28</code>
<code>Start_Oct05_MOF.py</code>	<code>d69d1f100b74ffa8bcc8eae2394e1bfd</code>	<code>4079c58306a2e15568bb2c006fa65eb5</code>

Injection technique: create a suspended `notepad.exe`, allocate RWX memory, write shellcode via `WriteProcessMemory`, queue an APC, resume the thread. The code is 151 lines, well-structured, with readable variable names and full ctypes structure definitions. A comment in the source reads: "CHANGE 2: Use the process argument from mof.py as the default here". This is development-stage code.

Type B: Python loaders (RC4 + junk code)

Two files (`Oct05_pyt.py`, `Start_Oct05_Python.py`), each ~820 KB. RC4 with 8-character keys:

File	RC4 Key
Oct05_pyt.py	xNuJ5Z75
Start_Oct05_Python.py	0ADNeZBG

Instead of process injection, these decrypt in-place and execute via `ctypes.CFUNCTYPE` -- shellcode runs directly in the Python process. Variable names are randomized 64-character strings with dead-code conditional blocks scattered throughout, but the RC4 implementation underneath is standard Key Scheduling Algorithm (KSA) + Pseudo-Random Generation Algorithm (PRGA).

The full chain (Wave 1)

```
Python (XOR or RC4)
-> Donut shellcode (Chaskey CTR, 16 rounds)
-> Fviwknzr.exe (.NET 3DES-CBC, 588,288 bytes)
-> Jaglt resource (AES-encrypted + GZip-compressed)
-> Qdjlj.dll (PureLogs, ConfuserEx-protected, protobuf C2)
```

Waves 2--3 (November 10) -- AES-256-CBC + dual XOR

14 plaintext Python loaders deployed across two staging directories (`3D0bj` for Wave 2/BKSNO, `3Dadu` for Wave 3/BKNO). All use an identical 172-line template. Python version: **3.12**.

The encryption scheme gets its first upgrade:

```
Encryption: XOR(key1) -> XOR(key2) -> AES-256-CBC(key, iv)
Decryption: base64_decode -> extract IV (first 16 bytes)
-> AES-256-CBC decrypt (PyCryptodome)
-> XOR(xor_key2) -> XOR(xor_key1) -> raw Donut shellcode
```

The injection method also changes. Instead of `notepad.exe`, the loaders now create a suspended `explorer.exe` and use Early Bird APC injection -- queuing the APC before the first instruction executes. Explorer blends in better than Notepad as a long-running process.

Seven RAT families deployed simultaneously per directory:

Loader	RAT	Shellcode Size
BKSNOLazyNov10_Anacrack.py	DcRat	103,186 B
BKSNOLazyNov10_Annorii.py	DcRat	103,186 B
BKSNOLazyNov10_Anoriihv.py	PureHVNC	101,138 B
BKSNOLazyNov10_Asyncccc.py	AsyncRAT	84,754 B
BKSNOLazyNov10_Plogggggg.py	PureLogs	629,522 B

Loader	RAT	Shellcode Size
BKSNOLazyNov10_Venommm.py	VenomRAT	103,186 B
BKSNOLazyNov10_Viooooooooo.py	Violet RAT	1,464,594 B

Wave 3 (BKSNOLazy) carries the same seven payloads re-encrypted with different keys. Shellcodes are byte-identical to Wave 2 -- redundant deployment to two directories.

Each loader has unique AES-256 keys, unique XOR keys, and unique IVs. All Wave 2/3 payloads are delivered through Donut v0.9.2 directly to .NET assemblies in memory. The scripts are still plaintext and easy to read. This is the transition stage before the Kramer shift.

Waves 4--5 (November 19) -- Kramer obfuscator

The biggest architectural shift in the campaign. Files are named .py but are compiled Python 3.12 bytecode (magic cb0d0d0a). The encryption simplifies back to RC4 with 8-character keys. Everything else gets dramatically more complex.

The Kramer class

Each file contains a single class definition instantiated at module scope:

```
Kramer(_kramer=False, _rasputin=False, _sparkle=<encoded_blob>)
```

The `__init__` method chains four lambda functions into a decode pipeline:

Lambda 1 (self._bit) -- hex decode. Imports `binascii` by reconstructing the module name from string indices ([1, 8, 13, 0, 18, 2, 8, 8] into a `_decode` string). Calls `unhexlify` on each /-separated segment of input. Returns decoded UTF-8 characters.

Lambda 2 (self._bits) -- chain function. Connects decode output to execution: calls `self._exit(Lambda3(input))`.

Lambda 3 -- anti-debug + triple decode.

1. Opens `__file__` and checks if the strings `print` or `input` appear anywhere in the source. Since the files are compiled bytecode, these strings are absent by default. If an analyst adds `print()` statements for debugging, the check triggers `exit()`.
2. **Layer 1:** Hex unhexlify via Lambda 1 -- produces Unicode codepoints
3. **Layer 2:** Character shift: `chr(ord(c) - SHIFT_CONSTANT)` for non-zeta characters; Greek zeta (U+03B6) maps to newline
4. **Layer 3:** Alphabet rotation: +1 position in `abcdefghijklmnopqrstuvwxyz0123456789` (a->b, z->0, 9->a)

Lambda 4 (self._exit) -- execution. Constructs the strings `exec`, `globals`, and `utf8` from index operations, then runs `eval(exec(globals(decoded_source)))`.

Polymorphic shift constants

Each sample uses a unique Unicode Private Use Area (PUA) shift constant, embedded deep in the bytecode constants of a nested generator expression. PUA codepoints (U+E000--U+F8FF) have no standard meaning and no displayable glyph -- they're invisible to signature-based detection that expects ASCII or common Unicode. Every file has a different constant, and the encoded blob is completely different even when the decoded payload is identical.

Set	File	RAT	Shift Constant	RC4 Key
EMP	1an-obf.py	VenomRAT	808,394	YMYpepLY
EMP	1as-obf.py	AsyncRAT	112,985	SgfdbVz0
EMP	1hv-obf.py	PureHVNC	196,742	zLning3R

Set	File	RAT	Shift Constant	RC4 Key
EMP	1plog-obf.py	PureLogs	774,771	teSGI2Zb
EMP	1vio-obf.py	Violet RAT	966,348	N0pzga4k
OBKS	OBKSLazyNov20_ancrack.py	VenomRAT	933,876	f9AibaSR
OBKS	OBKSLazyNov20_as.py	AsyncRAT	749,837	9V0000eh
OBKS	OBKSLazyNov20_hv.py	PureHVNC	362,709	5vCtSuqf
OBKS	OBKSLazyNov20_plog.py	PureLogs	671,200	ox1FPpGn
OBKS	OBKSLazyNov20_vio.py	Violet RAT	239,463	NQzn2pMo
WBKS	WBKSLazyNov20_ancrack.py	VenomRAT	529,419	u04dVbwt
WBKS	WBKSLazyNov20_as.py	AsyncRAT	556,625	LgQpDep7
WBKS	WBKSLazyNov20_hv.py	PureHVNC	441,206	7HJ4oP5i
WBKS	WBKSLazyNov20_plog.py	PureLogs	597,727	gkT854GZ
WBKS	WBKSLazyNov20_vio.py	Violet RAT	491,726	Hs3TFDx1

All 15 samples decode to the same Python template:

```
import ctypes
import base64

def rc4_decrypt(key, data):
    # Standard RC4 (KSA + PRGA)
    ...

def execute_shellcode():
    encrypted_data = base64.b64decode('<base64_blob>')
    key = '<RC4_KEY>'.encode('ascii')
    shellcode = rc4_decrypt(key, encrypted_data)

    shellcode_buffer = ctypes.create_string_buffer(shellcode)
    ctypes.windll.kernel32.VirtualProtect(
        ctypes.byref(shellcode_buffer),
        ctypes.sizeof(shellcode_buffer),
        0x40, # PAGE_EXECUTE_READWRITE
        ctypes.byref(ctypes.c_ulong())
    )
    shellcode_func = ctypes.cast(
        shellcode_buffer,
        ctypes.CFUNCTYPE(ctypes.c_void_p)
    )
    shellcode_func()

execute_shellcode()
```

The WBKS delta

Three build profiles exist: EMP, OBKS, and WBKS. EMP and OBKS carry identical shellcode sizes per RAT family but use different shift constants and RC4 keys -- the build system repackages the same shellcode with fresh obfuscation parameters.

WBKS shellcodes are consistently **+26,628 bytes** larger than EMP/OBKS across all five RAT families:

RAT	EMP/OBKS Shellcode	WBKS Shellcode	Delta
VenomRAT	360,702 B	387,330 B	+26,628
AsyncRAT	342,270 B	368,898 B	+26,628
PureHVNC	884,478 B	911,106 B	+26,628
PureLogs	887,038 B	913,666 B	+26,628
Violet RAT	1,722,110 B	1,748,738 B	+26,628

The constant delta across all five payloads strongly suggests WBKS adds an additional ~26 KB stage to the shellcode. We did not extract the WBKS-specific stub to confirm its purpose -- it could be an extra unpacking layer, an AMSI/ETW bypass, or a different Donut configuration -- but the exact +26,628 byte consistency across five unrelated RAT families rules out coincidence.

File size bloat

The obfuscation cost is significant. Nov10 plaintext loaders range from 119 KB to 1,959 KB. Nov19 Kramer loaders range from 4,126 KB to 20,684 KB. The Violet RAT loader -- the largest -- is 20.7 MB for a file that decodes to the same ~40-line RC4 template. That is a ~30x file size increase for the same functional payload.

The 7-layer chain (Nov19 only)

Wave 4/5 introduces the deepest nesting observed in the campaign. The Nov19 Donut instances deliver native x64 PE wrappers instead of .NET assemblies directly:

```
Layer 1: Python (.pyc disguised as .py)
Layer 2: Kramer decode (hex -> unicode shift -> rotation -> RC4 -> base64)
Layer 3: Donut #1 (Chaskey CTR, v0.9.3, 20,345-byte stub)
Layer 4: Native x64 AES crypter PE
Layer 5: AES-256-CBC + 16-byte XOR overlay
Layer 6: Donut #2 (Chaskey CTR)
Layer 7: .NET RAT assembly
```

The native AES crypter at Layer 4 uses per-sample keys stored in the `.rdata` section: 32-byte AES-256 key at offset `+0x210`, 16-byte CBC IV at `+0x230`, 16-byte XOR overlay key at `+0x240`. All Nov19 modules share an identical 176 KB `.text` section (the native loader stub) plus a `__sysc` section for direct syscalls.

Nov10 vs Nov19 comparison

Feature	Nov10 (plaintext)	Nov19 (Kramer)
File format	Python source (.py)	Python bytecode (.pyc as .py)
Obfuscation	None	Hex/Unicode/Caesar + anti-debug
File size range	119 KB -- 1,959 KB	4,126 KB -- 20,684 KB
Payload encryption	AES-256-CBC + 2x XOR	RC4 with 8-char key
Execution method	Early Bird APC into explorer.exe	In-process CFUNCTYPE
Anti-debug	None	Source string checks (<code>print/input</code>)

Feature	Nov10 (plaintext)	Nov19 (Kramer)
Build variants	1 set (7 payloads)	3 sets (15 files)
Key management	Fixed per payload	Unique per file (polymorphic)
Donut version	v0.9.2 (.NET direct)	v0.9.3 (native PE wrapper)

Donut v0.9.2 -- Chaskey CTR and AMSI bypass

Donut is the bridge between the Python shellcode and .NET. Every wave uses it. The framework packages .NET assemblies as position-independent shellcode that bootstraps the CLR from scratch.

Chaskey cipher

All Donut instances use the Chaskey block cipher -- a lightweight permutation-based MAC designed for microcontrollers (Mouha et al., 2015), repurposed by Donut as a stream cipher in CTR mode:

- **Block size:** 128 bits (16 bytes)
- **Rounds:** 16
- **Mode:** CTR with big-endian byte-array counter (incremented from byte[15] toward byte[0] with carry propagation)
- **Pre-whitening:** block XOR key
- **Post-whitening:** block XOR key

Instance layout

```
[+0x000] uint32   len           -- total instance size
[+0x004] byte[16] Chaskey key  -- 128-bit key
[+0x014] byte[16] Chaskey ctr  -- 128-bit counter/nonce
[+0x024] byte[4]  (zero pad)
[+0x028] uint64   API hash     -- VirtualAlloc
[+0x048] uint64   API hash     -- VirtualProtect
...
[+0x230] byte[]   Encrypted module data (v0.9.2)
[+0xC08] byte[]   Encrypted module data (v0.9.3)
```

AMSI/WLDP patching

Before loading the .NET assembly, every Donut instance runtime-patches five security scanning functions:

Function	DLL	Purpose
<code>AmsiInitialize</code>	amsi.dll	Disable AMSI startup
<code>AmsiScanBuffer</code>	amsi.dll	Bypass buffer scanning
<code>AmsiScanString</code>	amsi.dll	Bypass string scanning
<code>WldpQueryDynamicCodeTrust</code>	wldp.dll	Bypass Windows Lockdown Policy
<code>WldpIsClassInApprovedList</code>	wldp.dll	Bypass class approval checks

After patching, Donut loads `mscorlib.dll`, calls `CLRCreateInstance` to start the .NET CLR (v4.0.30319), and invokes `ExecuteInDefaultAppDomain` with the target class and method names stored in the instance.

v0.9.2 vs v0.9.3

Feature	v0.9.2 (Waves 1--3)	v0.9.3 (Waves 4--5)
Module type	.NET assembly directly	Native x64 PE wrapper
Architectures	x86 + x64	x64 only
Module offset	Instance + 0x230	Instance + 0xC08
Decoder stub (x86)	7,637 B	N/A
Decoder stub (x64)	23,925 B	20,345 B

The x64 decoder stub is 3x larger than x86 due to 64-bit register operations and wider instruction encoding.

Per-instance Chaskey keys (Oct05)

Instance	Chaskey Key	Counter	AppDomain
oct05_mof	63c77584066e65dc30f81dd14b782886	42342683636e4347129ce25e3ad4954b	T6X7NPPF
oct05_start_mof	b739245dfae51c8a399b7a189d8cf5c3	b961ea70e68d7751f371f9c307976343	FNCH7YMH
oct05_pyt	8fce7db93aba587407f368cb5c138ea1	bc71d7ebc0f2b0c512b5e8ff016b8840	FYXCHCX9
oct05_start_python	2cddb22a1c0ea2c59943e7841bfcc709	7f6ff3e61f9b57585fe17af90202c515	R9HRP33F

Batch stager infrastructure

The batch stagers are the initial execution layer. 29 .bat files recovered across six evidence directories deduplicate to 13 unique templates in four categories.

Download stagers

Two template families handle the initial download:

2Embammed -- AV-branched ZIP selection from a single Cloudflare tunnel per wave. Checks for AvastUI.exe and AVGUI.exe via tasklist. If detected, downloads abb11.zip (OBKS-only, Avast-safe profile). If not, downloads quz11.zip (full WBKS + BKSNO deployment). Each stager also downloads Shoopify.bat, PWS.vbs, and pws1.vbs into the Startup folder.

2bibo -- dual-ZIP download with fallback servers and random filenames. Downloads two ZIPs (Mfrac.zip and Sfrac.zip) into separate staging directories. Generates random 5-character uppercase filenames for local ZIPs. Has primary Cloudflare tunnel URLs plus fallback HTTP servers for redundancy.

Nine Cloudflare tunnels

Each deployment wave uses 2--3 disposable tunnels. A new tunnel is generated for each wave, with one exception: the Feb 1 Main_HV_x86 stager reuses the Dec 17 tunnel.

#	Tunnel	Stager	Date
1	mustang-allowing-them-legislative.trycloudflare.com	2Embammed_Stark	Nov 19

#	Tunnel	Stager	Date
2	valuation-throws-sixth-disc.trycloudflare.com	2Embambded_Main	Nov 28
3	brochure-pot-tested-clubs.trycloudflare.com	2bibo__Stark	Nov 28
4	candidates-burlington-hugh-anymore.trycloudflare.com	Main_HVNC_x86__Stark	Nov 28
5	render-seen-sensor-urban.trycloudflare.com	2Embambded_Main	Dec 17
6	avoiding-ended-holiday-encyclopedia.trycloudflare.com	2bibo__	Dec 17
7	blowing-paris-indoor-links.trycloudflare.com	Main_HV_x86	Dec 17
8	brunette-assembled-america-homepage.trycloudflare.com	2Embambded_Main	Feb 1
9	stomach-cite-personality-money.trycloudflare.com	2bibo__	Feb 1

Fallback servers

The 2bibo template includes non-Cloudflare fallback servers for when tunnels go down:

Server	Port	Used By
fsankmas.it.com	7333	Nov 28 Stark stager
fsankmas.xyz	7333	Nov 28 Stark stager
tammhdka.shop	5044	Dec 17 + Feb 1 stagers

RAT launcher evolution (Shoopify.bat)

Three versions of **Shoopify.bat** recovered, deployed to the Startup folder for persistence:

Version 1 (Nov 28): References Nov20 filenames for OBKS/WBKS, Sep07 filenames for BKSNO. No XWorm. AV detected: 5 loaders. No AV: 12 loaders.

Version 2 (Jan 7): References Dec16 filenames across all sets. No XWorm. Same loader counts.

Version 3 (Feb 1): Adds XWorm (**xwrm3**) to the OBKS and WBKS sets. AV detected: **6 loaders**. No AV: **13 loaders**. XWorm was never added to the BKSNO background set -- it was patched in between Jan 7 and Feb 1 by adding two lines to the foreground sets only.

This is the largest deployment profile in the campaign. When no Avast/AVG is detected, the system boots with 13 simultaneous Python RAT loaders, plus the Oct05 pair via **Kindle_x86.bat**, plus the Autolt chains, plus two anti-idle VBS scripts.

Anti-forensic evolution

Early stagers used simple **taskkill /f /im python.exe**. By Nov 28, they switched to WMI parent-process hunting:

```

Get-CimInstance Win32_Process -Filter "Name='explorer.exe'" |
ForEach-Object {
    $p = Get-Process -Id $_.ParentProcessId -ErrorAction Stop
    if ($p.ProcessName -eq 'python') {
        Stop-Process -Id $p.Id -Force
    }
}

```

The method is narrower: instead of killing all `python.exe` processes, it finds `explorer.exe` instances whose parent is `python.exe` and kills only that parent. `Kindle_x86.bat` extends this to also hunt `notepad.exe` parents, because Wave 1 MOF loaders inject into notepad.

Anti-idle scripts

Deployed by all download stagers to the Startup folder:

Script	Key	Interval
PWS.vbs	F15 (exists in USB HID spec, not on keyboards)	Every 55 seconds
pws1.vbs	Shift	Every 40 seconds

F15 resets the idle timer without producing visible keystrokes. Both scripts keep the session alive for PureHVNC remote access.

Encryption evolution summary

Generation	Waves	Date	Encryption	Injection	File Sizes
1	1	Oct 5	Double-XOR / RC4	notepad.exe APC / in-process	~820 KB
2	2–3	Nov 10	AES-256-CBC + dual XOR	explorer.exe Early Bird APC	119 KB -- 1,959 KB
3	4–5	Nov 19	Kramer .pyc + RC4	In-process CFUNCTYPE	4,126 KB -- 20,684 KB
4	6	Dec 17	Wave 1 redeployment	Same as Wave 1	Same as Wave 1
5	7	Dec 16	Mixed (all prior generations)	Mixed	Mixed

Progression summary: readable Python with two XOR keys -> AES-256 with PyCryptodome -> compiled bytecode with polymorphic Unicode PUA ciphertext and 30x file bloat. The RAT payload families stayed the same.

Indicators of compromise

Cloudflare tunnel domains

```

mustang-allowing-them-legislative.trycloudflare.com
valuation-throws-sixth-disc.trycloudflare.com
brochure-pot-tested-clubs.trycloudflare.com
candidates-burlington-hugh-anymore.trycloudflare.com
render-seen-sensor-urban.trycloudflare.com
avoiding-ended-holiday-encyclopedia.trycloudflare.com
blowing-paris-indoor-links.trycloudflare.com
brunette-assembled-america-homepage.trycloudflare.com
stomach-cite-personality-money.trycloudflare.com

```

Fallback servers

Domain	Port
fsankmas.it.com	7333
fsankmas.xyz	7333
tammhdka.shop	5044

Python loader hashes (Wave 1)

File	MD5
Oct05_MOF.py	f2eccf363c89098a445d3ed8a047c697
Oct05_pyt.py	ee6a79204c207b366e645f62ea493407
Start_Oct05_MOF.py	9277a1bd906839af54584d3ac6523caf
Start_Oct05_Python.py	2cd05753d1355e9e95c6001d7d2a64b7

Batch stager hashes

Template	MD5
2Embambded_Stark (Nov 19)	55aa4cb824722e9dd8e64d0538bd932b
2Embambded_Main (Nov 28)	4bf46fa39a5d9a9b94af8552d58bc974
2Embambded_Main (Dec 17)	891490ccab1f9eaa16a11be3330d60ee
2Embambded_Main (Feb 1)	89713e3f754164d104ea5ae4ee6e17fc
2bibo__Stark (Nov 28)	c8d885e9eb7a90228ebd5bf34afa49c6
2bibo__ (Dec 17)	8be09bb84bb3f2e8528862201420aa17
2bibo__ (Feb 1)	29f3b9286beecd8ff405222e182c4226
Shoopify (Nov 28)	aafd399291d36afb518d4cb22eceb616
Shoopify (Jan 7)	a9c46ef1a21a5799f608742e1a26ff43
Shoopify (Feb 1)	fd267fde4c228922c177fdda6e358b87

Behavioral indicators

Indicator	Context
python.exe spawning explorer.exe (suspended)	Early Bird APC injection (Waves 2--3)
python.exe spawning notepad.exe (suspended)	Process injection (Wave 1 MOF)
VirtualProtect with PAGE_EXECUTE_READWRITE from Python	In-process shellcode execution (Waves 1, 4--5)
AmsiInitialize / AmsiScanBuffer patching at runtime	Donut AMSI bypass (all waves)
WldpQueryDynamicCodeTrust patching	Donut WLDAP bypass (all waves)
WMI query killing python.exe parent of explorer.exe	Anti-forensic cleanup (Nov 28+)
F15 or Shift keypress every 40--55 seconds	Anti-idle keepalive (PWS.vbs / pws1.vbs)
attrib +h on %USERPROFILE%\Contacts	Hidden staging directory

Five posts, one kill chain

This post is the fifth and final entry in the SERPENTINE#CLOUD analysis series:

- 1. [PureLogs: Reverse Engineering a .NET RAT](#) -- the ConfuserEx-protected infostealer with protobuf C2
- 2. [PureCrypter: Reverse Engineering a .NET Loader](#) -- the 3DES event-driven pipeline
- 3. [Violet RAT v4.7: The Most Dangerous Payload in a 9-RAT Toolkit](#) -- 120-command dispatcher with ransomware and HVNC
- 4. [Remcos Banking Fraud via Three Autolt Persistence Chains](#) -- BYOI persistence and Canadian banking fraud
- 5. **Python Loader Evolution** (this post) -- the delivery pipeline

Most development effort went into the delivery pipeline: polymorphic build systems generating unique keys per file, AV-aware branching selecting deployment profiles at runtime, and disposable Cloudflare tunnels rotated by wave. The inner RATs -- DcRat, AsyncRAT, VenomRAT, Violet, and PureLogs -- ran on the same C2 infrastructure throughout.

The outer layers changed constantly. The inner layers barely changed at all.

Kirk

I like the internet. Want to get in touch? ·