Snakes in the Castle: Inside the Walls of Python-Driven CastleLoader Delivery

Archived: 2026-04-05 19:05:55 UTC

The Blackpoint SOC recently responded to an incident initiated through the tried-and-true ClickFix technique; a social engineering method consistently leveraged across numerous campaigns this past year. These lures convince users to press **Win + R** to open the Windows Run dialog box, then enter a command presented as a harmless “human verification” step or similar prompt. This pattern has been repeatedly used to deploy everything from information stealers to remote access trojans (RATs), and it has also become one of the primary delivery vectors for a newer loader family known as **CastleLoader**.

CastleLoader is a malware loader family that emerged in early 2025, primarily delivered through these same ClickFix campaigns. Its role is simple but dangerous, retrieve an encrypted second stage from attacker infrastructure, decrypt it in memory, and pass execution to whatever payload the operator chooses. Campaigns have leveraged CastleLoader to deploy a wide range of malware including remote access trojans like **CastleRAT**, **SectopRAT**, **NetSupport RAT**, and **WarmCookie**, along with credential stealing families such as **Stealc**, **RedLine**, **Rhadamanthys**, and **MonsterV2**.

In this case, the ClickFix command downloaded a small archive and staged its contents inside the user’s AppData directory before invoking a bundled copy of **pythonw.exe** to execute one of the extracted files. That script served as a simple Python stager whose only job was to rebuild and execute a CastleLoader payload. It decoded a Base64 encoded blob, applied a XOR routine using a hardcoded key, reconstructed the shellcode in memory, and transferred execution to it without ever writing an executable to disk.

After the shellcode took over, CastleLoader moved into its next phase of retrieving the attacker’s final payload. This stage leverages a technique known as **PEB Walking** to locate loaded modules and resolve all required APIs entirely at runtime, bypassing the normal Windows import system that security tools often watch closely. Once its function table was built, it contacted attacker infrastructure using a hardcoded **GoogeBot** User-Agent header, downloaded the final stage, decrypted it using the first 16 bytes of the blob as the XOR key, and executed it directly in memory.

Key Findings

- A ClickFix campaign leveraged a Python-based delivery to stage CastleLoader.
- Execution was proxied through conhost.exe, which invoked a cmd.exe downloader.
- The downloader fetched and unpacked a tar archive into an AppData staging folder.
- A windowless Python interpreter then ran python.cat, a compiled Python bytecode file.
- The Python stager decoded a Base64 blob and XOR-reconstructed the CastleLoader shellcode in memory.
- CastleLoader used PEB Walking and hashed API resolution to build its function table.
- The loader contacted attacker infrastructure using a GoogeBot User-Agent header.
- The final payload was downloaded, XOR-decoded using the first 16 bytes as the key, and executed in memory.

Observed Kill Chain

The infection chain began with a ClickFix style Run dialog command that the user was socially engineered into executing. This single line launched a hidden **conhost.exe** window, which proxied an execution chain of several built in Windows utilities. It first used **curl.exe** to download an archive originally named **p2.tar** from an attacker IP address, saving it as **pt.tar** in the user's **AppData\Roaming** directory. The command then created a new folder named **etc** and extracted the archive's contents into that location using the **tar** utility, all without displaying any visible window or prompt to the user.

Figure 1: Initial ClickFix command executed via conhost.exe.

Once the files were staged, the command invoked the bundled **pythonw.exe** inside the extracted folder. This is the windowless version of the Python interpreter, meaning it runs Python code without displaying a console window or any visible output to the user. The interpreter executed a file named **python.cat**, which is actually compiled Python bytecode. Bytecode is a lower-level representation of a Python script that the interpreter can run directly, and it doesn't contain readable source code, making the program logic opaque without decompiling it.

Figure 2: Confirming python.cat is compiled Python bytecode.

To analyze the loader's behavior, the compiled bytecode needs to be decompiled back into a readable form. Python bytecode can't be understood directly, but it can be translated back into approximate source code using decompilation tools. A tool such as **Decompyle++** can take the bytecode from **python.cat** and reconstruct the script logic well enough to review its flow and understand the functions implemented by the attacker.

Figure 3: Recovering readable script logic from python.cat using Decompyle++.

With the bytecode decompiled, the recovered script reveals a compact loader that hides its real logic behind a block of text mixing normal Latin letters with Cyrillic characters. The script runs this encoded string through a series of **.replace()** calls, swapping each Cyrillic glyph back to another Latin character or digit until the text becomes a valid Base64 string. After reconstructing the clean Base64 data, the script decodes it with **base64.b64decode()** and immediately executes the result with **exec()**, meaning the visible script is just a wrapper, and the actual loader logic lives inside the decoded payload.

Figure 4: Loader reconstructing the payload via Cyrillic replacements.

After decoding the obfuscated wrapper, the resulting Python script is a straightforward in-memory shellcode loader. It defines a **xor_decrypt()** function that takes two-byte arrays, loops over each byte of the encrypted payload, and XORs it with a repeating key. The script then Base64 decodes two hardcoded blobs, the first of which is encrypted shellcode, and the second is the key used to decrypt it. Passing both into **xor_decrypt()** produces a decrypted shellcode buffer, which is wrapped in a **bytearray** for further use.

The rest of the script uses **ctypes** to pass the decrypted bytes directly into Windows' low level memory routines. It calls **VirtualAlloc** to create a new memory region with read, write, and execute permissions, large enough to hold the shellcode. After that, it builds a C-style buffer containing the decrypted bytes and uses **RtlMoveMemory** to copy them into the allocated region, a common technique used by in-memory loaders. Finally, the script uses **CFUNCTYPE** to create a function pointer and cast the allocated memory address to it, allowing Python to treat that region as a callable function. The script calls the function pointer, completing the transfer of execution into the decoded payload running inside the **pythonw.exe** process.

Figure 5: The decoded Python loader decrypting and executing shellcode in memory.

To examine the next stage of the kill chain, the loader's decryption logic can be reused outside of the **pythonw.exe** process. By taking the same Base64 blobs and XOR routine shown earlier and removing the in-memory execution step, the script can be made to simply write the decrypted bytes to disk instead. This produces a standalone shellcode sample that can be analyzed further to understand its behavior.

Figure 6: Modified script that decrypts the shellcode and writes it to shellcode.bin.

Loading the decrypted shellcode into a disassembler immediately reveals that it targets a 32-bit x86 environment. The entry routine uses the classic 32-bit function prologue (**push ebp** followed by **mov ebp, esp**) and works entirely with registers like **eax**, **ecx**, and **edx**, with stack variables referenced using 4-byte offsets. These registers belong to the 32-bit register set,

which indicates that the shellcode is built for an x86 execution context. This is a common choice for loader malware, since 32-bit shellcode runs consistently on modern 64-bit systems under WoW64 and lets attackers rely on simpler, well tested methods.

Figure 7: Decompiled routine showing 32-bit register usage and stack-based operations.

After cleaning up the decompiled shellcode into a more readable form by renaming functions and variables, the execution flow becomes straightforward to follow. The primary function we'll refer to as **loaderEntry()** builds the hardcoded domain **dperforms[.]info**, appends the path **/service/download/load_1**, and passes the resulting URL to a function renamed to **httpDownloadPayload()**. The server's response is stored in variables renamed as **responseBuf** and **responseSize**.

Figure 8: loaderEntry constructing the hardcoded staging URL and initiating the download.

Examining the **httpDownloadPayload()** function to better understand the network request reveals it serves as a lightweight HTTP client that resolves WinINet APIs at runtime, issues the outbound request, and streams the response into a heap buffer. The routine also writes the hardcoded string **GoogeBot** into a small buffer used as the **User-Agent** header. Based on the parameters passed in from **loaderEntry()** such as the destination buffer, the size pointer, and the full staging URL, the function initializes the request with those values and returns the downloaded contents through **outBuf** and **outSize**.

Figure 9: GoogeBot User-Agent and WinINet setup in the httpDownloadPayload function.

There are two distinct overlaps in this stage matching indicators observed in **CastleLoader** campaigns. The hardcoded User-Agent string **GoogeBot** has appeared in prior CastleLoader traffic, where it is used to disguise outbound requests with a crawler style identifier. The staging path assembled by the loader, which includes **/service/download/**, is another pattern that has been repeatedly observed in **CastleLoader** network requests. Both traits are well known **CastleLoader** markers, and their presence in this Python based chain is not coincidental.

CastleLoader is a newer loader family that emerged in early 2025, most commonly delivered through ClickFix style Run dialog social engineering. Its job is to retrieve an encrypted second stage from attacker infrastructure, decrypt it in memory, and pass execution to whatever payload the operator chooses. Campaigns using this tooling have historically deployed a broad mix of malware, including remote access tools like **NetSupport RAT**, **SectopRAT**, **CastleRAT**, and **WarmCookie**, as well as information stealers such as **StealC**, **RedLine**, **Rhadamanthys**, **DeerStealer**, **MonsterV2**, and others.

Moving back to the shellcode, one of its core behaviors is the use of a technique known as **PEB Walking** to resolve every API it needs at runtime. Typically, programs list their Windows API calls in an **Import Table** and the OS loads them automatically. CastleLoader avoids that by reading the **Process Environment Block (PEB)**, iterating through the modules already loaded in memory, and manually locating function addresses by hashing each exported name until a match is found. This allows the shellcode to retrieve memory allocation routines, networking functions, and other capabilities without relying on static imports.

The **PEB Walking** implementation relies on two helper functions. The first, which was renamed to **getModuleBase()**, works by selecting a DLL name based on an integer identifier, constructing that name in memory, and then scanning the PEB's module list to locate the corresponding loaded module. This gives the shellcode direct access to the base address of libraries like **ntdll.dll**, **kernel32.dll**, and **wininet.dll** without relying on a traditional import table.

Figure 10: The getModuleBase() function locating DLLs by walking the PEB module list.

The second function, renamed to **resolveApiByHash()**, takes that module base and walks the module's Export Table, hashing each exported name and comparing it to constants embedded in the loader. When a match is found, it returns the function's address as a callable pointer. Used in tandem, the two functions enable the shellcode's **PEB Walking** logic, where **getModuleBase()** retrieves the base address of a target DLL and **resolveApiByHash()** uses that base to resolve individual API functions dynamically.

Figure 11: resolveApiByHash() parsing export entries and matching hashed names.

After the staging request returns to **loaderEntry()**, the downloaded blob is split into two sections rather than being treated as a single buffer. The first 16 bytes at **buf** become the XOR key, and the remaining bytes form the encrypted payload beginning at **buf + 0x10**. Those values, along with the adjusted length **size – 0x10**, are passed into a function renamed to **xorDecrypt()**, which applies the key across the encrypted region and writes the decoded bytes back into the same memory. Once the loop completes, the decrypted payload is fully reconstructed in place at **buf + 0x10**, ready for execution.

Figure 12: Decryption step in loaderEntry() using the first 16 bytes as the key.

With the payload decrypted and staged in memory, the loader simply hands execution to the function pointer built from **buf + 0x10**. This is the point where control shifts entirely into whatever final stage the operator intended to deliver. Unfortunately, by the time this sample was analyzed, the staging domain was no longer returning a payload, leaving the actual final malware unknown. The final payload would have been decoded and executed directly in memory, consistent with the behavior of the rest of the chain.

Methodology & Attribution

The CastleLoader attribution is backed by two very specific indicators observed in the loader's HTTP request behavior. The hardcoded **GoogeBot** User-Agent directly overlaps with both GoogleBot and GoogeBot strings observed in

multiple CastleLoader campaigns throughout 2025. The staging URL assembled by the loader also contains the **/service/download/** path, a distinctive pattern that appears across several confirmed CastleLoader samples. Seeing both traits in the same request creates a strong match with known CastleLoader infrastructure and tooling.

The delivery method further supports this alignment. Earlier CastleLoader campaigns leveraged ClickFix techniques that dropped a ZIP archive containing AutoIt scripts, which in turn loaded the CastleLoader shellcode into memory. In this infection chain, the AutoIt loader has simply been replaced with a small Python dropper that writes and executes the shellcode, but the overall delivery concept remains the same. The delivery technique here matches the same approach CastleLoader has used across earlier samples.

Finally, the loader's behavior follows the same pattern. Historical CastleLoader samples relied on hashed DLL names, hashed API identifiers, and PEB Walking for export resolution, the same approach taken by this shellcode. Furthermore, while earlier CastleLoader samples relied on a static hardcoded key to decrypt the final payload, this variant slightly modifes the technique by deriving the key from the first 16 bytes of the downloaded blob. Even with that deviation, the core design is unchanged. The loader still retrieves an encrypted payload, decrypts it in memory, and stages it for execution, following the same overall pattern seen in earlier CastleLoader samples.

In conclusion, each stage of this chain lines up cleanly with known CastleLoader behavior, from the ClickFix delivery, to the distinctive GoogeBot user agent, staging path, and loader design. The Python dropper simply replaces the AutoIt stagers seen in earlier campaigns, but the shellcode it delivers follows the same architectural patterns, API resolution techniques, and execution flows seen previously. This combination of traits firmly aligns the activity with the CastleLoader family, delivered through a modified implementation of its rapidly evolving delivery technique.

Recommendations

- Educate end users on ClickFix style social engineering, especially lures that prompt them to open the Run dialog and enter "verification" commands.
- Use Group Policy to restrict or disable access to the Run dialog for users who do not require it operationally.
- Restrict access to cmd.exe, PowerShell, and Python runtimes for users who do not need them as part of normal workflow.
- Monitor or alert on suspicious LOLBin sequences, such as conhost.exe spawning cmd.exe, followed by curl, tar, or pythonw.exe.
- Implement DNS monitoring and filtering to block newly registered domains or low reputation TLDs commonly used for malware staging.
- Monitor for Python binaries executed from atypical directories, such as AppData, where legitimate Python installations are not typically located.
- Monitor and audit outbound network connections from pythonw.exe, since its windowless execution is commonly abused for stealthy staging activity.

Indicators of Compromise (IOCs)

Network

Type	Indicator	Context / Notes
IP	78.153.155[.]131	Stage 1 Delivery
Domain	dperforms[.]info	Final Stage Domain
URL	hxxp[://]78.153.155[.]131/service/download/p2.tar	Malicious Tarball
URL	hxxps[://]dperforms[.]info/service/download/load_1	Final Stage Payload

Files

Filename	Hash	Context / Not
pt.tar	8A539355D317BD8A490F470319410E5D2A2851A38828C900F357FBAC9083583C	Malicious Tar
python.cat	0F5C3AC4B4F997ACD2CD71C451082CD8FBD1CBDB1A6DB2BDF470714F2E7EF4BB	Python Stager
N/A	BFEA06A7EF5B25B40178CFFFD802D8AB4F5EE35CA5CD8D2B9FF29B4E201B3B7F	CastleLoader

---