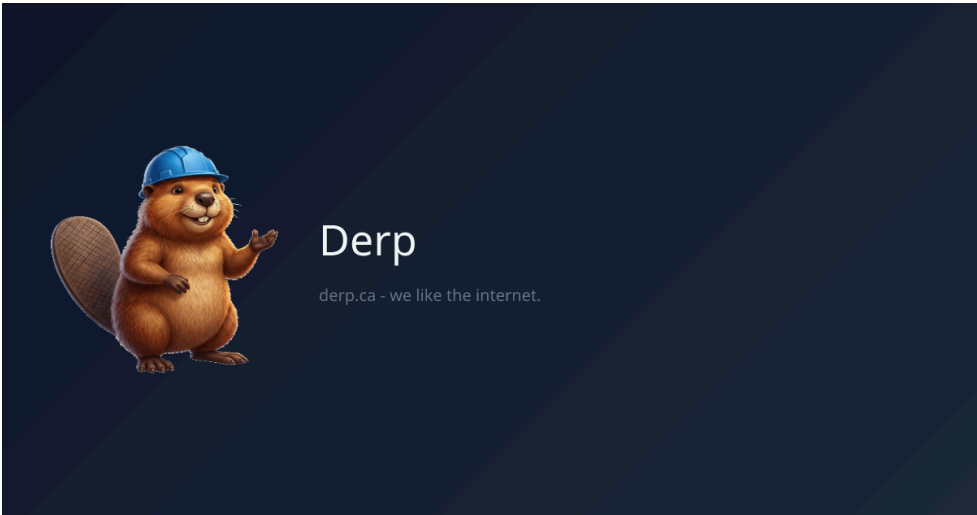


PureCrypter: Reverse Engineering a .NET Loader From the PureCoder Ecosystem

derp.ca/research/purecrypter-loader-analysis

Kirk

February 22, 2026



malware.netpurecrypterpurecoderincident-response

We recovered two ConfuserEx-obfuscated .NET executables from a multi-stage intrusion. They are builder-generated loaders -- their only purpose is to decrypt an embedded resource, decompress it, and load the result as a .NET assembly via reflection.

As of 2026-02-26, the Sep loader (**Fviwnzr.exe**) is 49/76 on VirusTotal, and the two Nov loader builds (**Erqcke.exe**) are both 41/76. The Oct05 and Nov loader hashes are now all present in Tria.ge with completed reports.

Both loaders are **PureCrypter**, the commercial .NET crypter sold as part of the **PureCoder** malware-as-a-service ecosystem. The campaign has been publicly tracked by Securonix as **SERPENTINE#CLOUD**. This analysis focuses on the outer wrapper. The inner payload delivered by one of these loaders is analyzed in [PureLogs: Reverse Engineering a .NET RAT](#).

Tria.ge and VT refresh (2026-02-26)

Artifact	SHA256	VT	VT first seen (UTC)	Tria.ge ID	Score
Fviwnzr.exe (Sep/Oct05 loader)	dcd22d338a0bc4cf615d126b84dfcda149a34cf18bc43b85f16142dfb019e608	49/76	2025-09-18 08:09:30	260226-na81hacs3e	7
Erqcke.exe (Nov10 loader)	0ab09a4787ea9cb259cadd3f811a56f7bd0058287634bbaf0388b2cd40464505	41/76	2026-02-22 01:35:20	260226-mt3gdsbx4g	3
Erqcke.exe (Nov19 loader)	b1c6659ee4ee35540f5ed043b611ac88a7fce9dc2f564168e7d47c43683163f6	41/76	2026-02-22 01:36:27	260226-mt3r6abx4h	3
Qdjjj.dll (Oct05 inner payload)	cdf87d68885caa3e94713ded9dd5e51c39b7bc7ef9bf7d63a4ff5ab917a96b36	17/76	2026-02-26 10:43:37	260226-mszzwabx2c	3

Artifact	SHA256	VT	VT first seen (UTC)	Tria.ge ID	Score
Mvfxsog.dll (Plog inner payload)	046d0e83c1e6dcdf526127b81b962042e495f5ae3a748f3a9452be62f905acf8	37/76	2026-02-22 01:33:52	260226-mszn4sxbx2b	3

The two Erqcke loader hashes remain low-score unsigned PE samples in Tria.ge despite 41/76 VirusTotal detection. The Oct05 loader scored higher in Tria.ge because detonation reached behaviour associated with the inner payload stage (browser profile reads and hook-related API use). The inner payload DLLs stayed score 3 with unsigned-PE-only static signatures in this refresh run.

The breach timeline also confirms cross-wave reuse: the Nov19 phvnc slot reused this exact Oct05 loader hash (dcd22d...) byte-for-byte.

Sample overview

Field	Fviwknzr.exe (Sep build)	Erqcke.exe (Nov10/Nov19 builds)
SHA256	dcd22d338a0bc4cf615d126b84dfcda149a34cf18bc43b85f16142dfb019e608	0ab09a4787ea9cb259cadd3f811a56f7bd0058287 (Nov10), b1c6659ee4ee35540f5ed043b611ac88a7fce9dc2 (Nov19)
Size	588,288 bytes	597,288 bytes (Nov10), 590,848 bytes (Nov19)
Type	PE32 .NET (x86), CLR v4.0.30319	PE32 .NET (x86), CLR v4.0.30319
PE timestamp	2025-09-04	2025-11-xx
Obfuscator	ConfuserEx v1.x	ConfuserEx v1.x
VT first seen	2025-09-18 08:09:30 UTC	2026-02-22 01:35:20 UTC (Nov10), 2026-02-22
VT detections	49/76 (trojan.msil/androm)	41/76 on both hashes (trojan.msil/jalapeno)
3DES key (base64)	Aab+zuk+b7ZNNoojdAhx7w==	KxqBSIKa5EkFaAuH0P+Png==
3DES IV (base64)	226dKBKRPbM=	X0W98LRYqZg=
Resource name	Jaglt (555,016 bytes)	Ctjady (555,704 bytes)
Inner payload	Qdjli.dll (PureLogs fat client)	Mvfxsog.dll (PureLogs plugin stager)
Entry class	bvvHVYU940gLdcXoTjh.jp8DQgU0wWFLN2sjUR8	Y89SbSJ6S8v0gcIg63.EL2VsZRu4no3pyibCR
Entry method	DYDUwZw01R	zoPKRGIHS
Namespace	Fviwknzr.DataProcessing	Erqcke.EventManagement

Field	Fviwknzr.exe (Sep build)	Erqcke.exe (Nov10/Nov19 builds)
Classes	14 (+ Module + AssemblyInfo)	15 (+ Module + AssemblyInfo)

Both loaders deliver structurally different PureLogs variants to the same campaign infrastructure. The inner payloads are analyzed separately -- this post covers only the loader framework.

Architecture: event-driven pipeline

Both loaders use the same architecture. An orchestrator class wires five event handlers into a chain. Each handler does one thing, fires the next event, and the pipeline cascades from start to finish:

```

Main()
-> new Orchestrator()
  -> constructor wires 5 event handlers:
    DecryptRequested -> 3DES-CBC decrypt
    Decrypted        -> GZip decompress
    Decompressed     -> Assembly.Load
    AssemblyLoaded   -> Reflection setup
    InvokeRequested  -> method.Invoke
-> Orchestrator.Start(resourceBytes, key, IV, className, methodName)
-> fires DecryptRequested
  -> fires Decrypted
    -> fires Decompressed
      -> fires AssemblyLoaded
        -> fires InvokeRequested
          -> done

```

Each stage fires a **StageChanged** event with a string label: **"Start"**, **"Decrypt"**, **"Decompress"**, **"AssemblyLoad"**, **"Reflect"**, **"Complete"**. Errors at any stage fire **ErrorOccurred** and re-throw.

Here is **Main()** from the Sep build (Fviwknzr.exe), deobfuscated:

```

// Fviwknzr.Jfvihq1.Main() -- deobfuscated
public static void Main()
{
    Orchestrator orch = new Orchestrator();
    try
    {
        orch.Start(
            Resources.Jaglt, // encrypted resource bytes
            Convert.FromBase64String("Aab+zuk+b7ZNNoojdAhx7w=="), // 3DES key
            Convert.FromBase64String("226dKBKRPbM="), // 3DES IV
            "bvvHvYU940gLdcXoTjh.jp8DQgU0wwWFLN2sjUR8", // target class
            "DYDUwZw01R" // target method
        );
    }
    catch { }
}

```

The Nov build (**Erqcke.exe**) is identical in structure with different randomized names and keys:

```
// Erqcke.Vavbij.Main() -- deobfuscated
public static void Main()
{
    Orchestrator orch = new Orchestrator();
    try
    {
        orch.Start(
            Resources.Ctjady,
            Convert.FromBase64String("KxqBSIkA5EkfaAuH0P+Png=="),
            Convert.FromBase64String("X0W98LRYqZg="),
            "Y89SbSJ6S8v0gcIg63.EL2VsZRu4no3pyibCR",
            "zoPKRGIHs"
        );
    }
    catch { }
}
```

The silent `catch { }` is characteristic of PureCrypter -- if anything fails, the process exits silently. No error logging, no crash dialog.

3DES decryption

The first stage handler decrypts the embedded resource using 3DES-CBC with PKCS7 padding. The key and IV are base64-encoded strings stored as `static readonly byte[]` fields in a dedicated key-store class.

Deobfuscated decrypt method (Sep build):

```
// Fviwknzr.Extraction.AccessibleExtractor -- deobfuscated
internal static byte[] Decrypt(byte[] data, byte[] key, byte[] iv)
{
    // key must be 16 or 24 bytes, IV must be 8 bytes
    using var tdes = new TripleDESCryptoServiceProvider();
    tdes.Mode = CipherMode.CBC;
    tdes.Padding = PaddingMode.PKCS7;
    tdes.Key = key;
    tdes.IV = iv;

    using var ms = new MemoryStream();
    using var cs = new CryptoStream(ms, tdes.CreateDecryptor(), CryptoStreamMode.Write);
    cs.Write(data, 0, data.Length);
    cs.FlushFinalBlock();
    return ms.ToArray();
}
```

The Nov build adds key zeroing after use (`Array.Clear(key, 0, key.Length)`) and descriptive error messages ("IV must be 8 bytes for 3DES", "Key must be 16 or 24 bytes"). The Sep build throws bare `ArgumentException()` with no message.

Extracted keys

Build	Key (base64)	Key (hex)	IV (base64)	IV (hex)
Sep (Fviwknzr)	Aab+zuk+b7ZNNoojdAhx7w==	01a6fecee93e6fb64d368a23740871ef	226dKBKRPbM=	db6e9d2812913db3
Nov (Erqcke)	KxqBSIkA5EkfaAuH0P+Png==	2b1a81488900e4491f680b87d0ff8f9e	X0W98LRYqZg=	5ce5bdf0b458a998

Both keys are 16 bytes (128-bit 3DES). The builder generates random keys and IVs for each build.

GZip decompression

After decryption, the result is GZip-compressed with a 4-byte little-endian size header prepended. The decompressor reads the expected size, decompresses, and validates the output length matches.

Deobfuscated (Sep build):

```
// Fviwnzr.DataProcessing.FilterEncryptor -- deobfuscated
internal static byte[] Decompress(byte[] data, int headerSize, int maxSize)
{
    // headerSize = 4, maxSize = 16777216 (16 MB)
    int expectedSize = BitConverter.ToInt32(data, 0);

    if (maxSize > 0 && expectedSize > maxSize)
        throw new InvalidDataException();

    using var stream = new MemoryStream(data, headerSize, data.Length - headerSize);
    using var gzip = new GZipStream(stream, CompressionMode.Decompress);
    using var output = new MemoryStream(expectedSize > 0 ? expectedSize : 0);

    gzip.CopyTo(output);
    byte[] result = output.ToArray();

    if (result.Length != expectedSize)
        throw new InvalidDataException();

    return result;
}
```

The Nov build replaces the `CopyTo()` call with a manual 4KB-buffer read loop (for .NET 3.5 compatibility) and adds a 100 MB hard limit on decompressed size before allocation.

Assembly loading and reflection invoke

The decompressed bytes are a .NET assembly. The loader calls `Assembly.Load(byte[])`, resolves the target type and method by name, and invokes it.

Sep build (deobfuscated):

```
// Assembly.Load stage
Assembly asm = Assembly.Load(decompressedBytes);

// Reflection setup stage
MethodInfo method = asm
    .GetType("bvvHVYU940gLdcXoTjh.jp8DQgU0wWFLN2sjUR8", throwOnError: true)
    .GetMethod("DYDUwZw01R", BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic);

// Invoke stage
method.Invoke(null, null); // no instance, no parameters
```

Nov build (deobfuscated):

```
Assembly asm = Assembly.Load(decompressedBytes);

Type type = asm.GetType("Y89SbSJ6S8vOgcIg63.EL2VsZRu4no3pyibCR", throwOnError: true);
MethodInfo method = type.GetMethod("zoPKRGIHs",
    BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public | BindingFlags.NonPublic);

object target = method.IsStatic ? null : Activator.CreateInstance(type);
method.Invoke(target, args); // can pass parameters to inner assembly
```

Two differences. The Nov build supports both static and instance methods (it creates an instance with `Activator.CreateInstance` if the method is not static). It also passes an `object[]` argument array instead of `null` -- the builder can bake arguments into the loader that get forwarded to the inner assembly. The Sep build only supports static methods with no arguments.

The Nov build also validates the decompressed bytes before loading -- it checks for the MZ header (`bytes[0] == 0x4D && bytes[1] == 0x5A`) and a minimum length of 64 bytes.

Obfuscation

Both builds are protected with ConfuserEx. The obfuscation is aggressive-looking but functionally trivial once you understand the pattern.

Control flow flattening

Every method in both assemblies uses the same structure:

```
int num = <initial>;
while (true)
{
    switch (num)
    {
    case 0:
        // some operation
        num = 3;
        break;
    case 1:
        return result;
    case 2:
        // another operation
        num = 0;
        break;
    // ...
    }
}
```

Linear code is broken into blocks, each assigned a case number. The `while(true) { switch }` dispatcher jumps between blocks by setting `num` to the next case. The original control flow is straightforward -- you just need to trace the case transitions to reconstruct it. Every method in this post has been through that reconstruction.

Opaque predicates

The `<Module>` class in each assembly contains over 100 `int` fields. These are initialized in a flattened static constructor via XOR expressions that mostly evaluate to 0:

```
// From <Module>{47d0803d-c8a4-4c82-9451-bbb9c835a0a8}
m_0529e0fe017441aabc8f0cb2aa9b130 = 0x6CBF766 ^ 0x6CBF766; // = 0
m_0991f046cea44229a53b00bd4b1e1bef = 0x4F4DEAD1 ^ 0x4F4DEAD1; // = 0
// ... 100+ more, all zero
```

These fields are referenced throughout the code in dead branches:

```
int num = 0;
if (<Module>.m_0529e0fe... != 0) // always false
{
    num = 0; // dead code
}
switch (num) // always 0
{
    // ...
}
```

The predicates never change the program's behavior. They exist to break static analysis tools that cannot resolve the field values. Once you know every predicate field is 0, the dead branches collapse and the real logic emerges.

Dead code injection

Beyond opaque predicates, ConfuserEx injects `if (field == 0) { num = 0; }` checks between every functional block. These are scattered throughout all methods in both builds. They are pure noise -- each one assigns the same value the variable already holds.

Builder evolution

The Nov Erqcke branch is a newer version of the PureCrypter builder. The differences below show the builder's development between September and November 2025:

Feature	Sep build (Fviwknzr)	Nov build (Erqcke)
Error messages	Bare <code>ArgumentException()</code>	Descriptive: "IV must be 8 bytes for 3DES", "Key must be 16 or 24 bytes", "Encrypted data cannot be null or empty"
Key zeroing	None	<code>Array.Clear(key, 0, key.Length)</code> after use
Size validation	None	100 MB max decompression limit, MZ header check
Stream copy	<code>GZipStream.CopyTo()</code> (.NET 4.0+)	Manual 4KB buffer loop (.NET 3.5 compatible)
Invoke arguments	<code>method.Invoke(null, null)</code>	<code>method.Invoke(target, args)</code> -- supports parameters
Method binding	Static only	Static + instance (Activator.CreateInstance fallback)
Error context	Exception only	Exception + stage name string
Class count	14	15 (extra stream utility class)

The builder generates all class names, method names, field names, and namespace names. Even the sub-namespace naming convention uses "real-sounding" prefixes (`.DataProcessing`, `.DecisionMaking` vs `.EventManagement`, `.Management`, `.Values`) -- these are also builder-generated randomized strings.

Both Nov hashes (`0ab09a47...`, `b1c6659e...`) map to this Erqcke branch. They are not byte-identical but they preserve the same class layout, decrypt-decompress-reflect pipeline, and 3DES material.

Class mapping

Every class in both builds has a direct functional equivalent with a different randomized name. Here is the complete mapping:

Role	Fviwknzr.exe (Sep)	Erqcke.exe (Nov)
Entry point	<code>Jfvihql.Main()</code>	<code>Vavbij.Main()</code>
Orchestrator	<code>Vmtxjyn</code> (1,060 lines)	<code>Ggrbywy</code> (1,115 lines)
3DES key store	<code>SetDecider</code>	<code>ElementEvent</code>
3DES decryptor	<code>AccessibleExtractor</code>	<code>ConvertiblePool</code>
GZip decompressor	<code>FilterEncryptor</code>	<code>InitializerVisitor</code>
Resource accessor	<code>Brldzgjjnp</code> -> <code>Jaglt</code>	<code>Wtduo</code> -> <code>Ctjady</code>
Stream utility	<code>(none)</code>	<code>LiteralValue</code>

Role	Fviwknzr.exe (Sep)	Erqcke.exe (Nov)
StageChanged args	Bygcocdv dg (string)	Edpsyp (string)
DecryptRequested args	Mokubjtf (byte[] x3)	Aepjgfg (byte[] x3)
Decrypted args	Gcbnen (byte[])	Nhdzdtuhc (byte[])
Decompressed args	Dkjwlljntsb (byte[])	Eqsknvcizh (byte[])
AssemblyLoaded args	Tgfhhk n (Assembly)	Rgypqjcdloq (Assembly)
InvokeRequested args	Mxrkricez (Asm, str, str)	Jdlwwh (Asm, str, str, obj[])
ErrorOccurred args	Tkiist (Exception)	Yqsmuahmhf (Exception, string)

The Nov build adds `LiteralValue`, a stream utility class providing a manual copy loop and a safe-dispose extension method. The Sep build uses `GZipStream.CopyTo()` directly and has no equivalent class.

Indicators of compromise

Loader hashes

Sample	SHA256	MD5	VT
Fviwknzr.exe (Sep/Oct05 loader)	dcd22d338a0bc4cf615d126b84dfcda149a34cf18bc43b85f16142dfb019e608	505a7b3074c5a769d85087b7f2e4bddd	49/
Erqcke.exe (Nov10 loader)	0ab09a4787ea9cb259cadd3f811a56f7bd0058287634bbaf0388b2cd40464505	3954ddd9e691ab0520e82f83d53579d4	41/
Erqcke.exe (Nov19 loader)	b1c6659ee4ee35540f5ed043b611ac88a7fce9dc2f564168e7d47c43683163f6	e3d3cfdde5349c2728b1b7a0c9ccfdc1	41/

3DES parameters

Build	Key (base64)	IV (base64)	Resource
Sep (Fviwknzr)	Aab+zuk+b7ZNNoojdAhx7w==	226dKBKRPbM=	Jaglt
Nov (Erqcke)	KxqBSIkA5EkfaAuH0P+Png==	X0W98LRYqZg=	Ctjady

Reflection targets

Build	Entry class	Entry method
Sep	bvvHVYU940gLdcXoTjh.jp8DQgU0wWFLN2sjUR8	DYDUwZw01R

Build	Entry class	Entry method
Nov	Y89SbSJ6S8vOgcIg63.EL2VsZRu4no3pyibCR	zoPKRGIHs

Inner payloads

Loader	Inner assembly	SHA256	VT	Tria.ge
Fviwknzr.exe	Qdjij.dll (1,290,752 bytes)	cdf87d68885caa3e94713ded9dd5e51c39b7bc7ef9bf7d63a4ff5ab917a96b36	17/76	260226-mszwwabx2c (score 3)
Erqcke.exe	Mvfxsog.dll (1,319,424 bytes)	046d0e83c1e6dcdf526127b81b962042e495f5ae3a748f3a9452be62f905acf8	37/76	260226-mszn4sbx2b (score 3)

Behavioral indicators

- .NET process loading an embedded resource, decrypting with 3DES-CBC, and decompressing with GZip
- 4-byte little-endian size header prepended to GZip stream
- In-memory .NET assembly loading via `Assembly.Load(byte[])`
- Reflection invocation of a specific class/method in the loaded assembly
- ConfuserEx control flow flattening (`while(true) { switch }` pattern)
- Module-level opaque predicate fields (100+ `int` fields all evaluating to 0)

Connection to PureLogs

Both loaders deliver PureLogs variants. The inner payload from Erqcke.exe (Mvfxsog.dll) is the subject of [PureLogs: Reverse Engineering a .NET RAT](#) -- a plugin stager with no built-in offensive capability that waits for .NET assemblies pushed from C2.

The inner payload from Fviwknzr.exe (Qdjij.dll) is a fat client -- a monolithic build with crypto wallet theft, browser credential harvesting, Telegram/Foxmail data theft, and PowerShell persistence baked in. Same ecosystem, different operational profiles.

The loader coverage now spans Sep, Nov10, and Nov19 hashes. `Fviwknzr.exe` remains the Oct05 loader branch, and both Nov hashes are Erqcke branch recompiles using the same Ctjady -> Mvfxsog handoff pattern. The loader stage changes names, hashes, and wrappers across waves, but the 3DES -> GZip -> Assembly.Load -> reflection pattern stays stable.

PureCrypter is the delivery mechanism. PureLogs is the payload. Both are sold as part of PureCoder's malware-as-a-service toolkit alongside PureHVNC and repackaged commodity RATs (AsyncRAT, VenomRAT, DcRat, XWorm). The Autolt-based persistence layer that kept these loaders alive across reboots is analyzed in [Remcos Banking Fraud & Autolt Persistence](#). The campaign was first publicly documented by Securonix as SERPENTINE#CLOUD.

Kirk

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