Детектирование вредоносной десериализации в CLR (CVE-2025-53770)

Александр Родченко

Летом 2025 года злоумышленники атаковали по всему миру сервера SharePoint, получая полный контроль над ними с помощью цепочки эксплоитов ToolShell. Эксплуатируются при этом уязвимости CVE-2025-49704 и CVE-2025-49706, а также их исправленные версии CVE-2025-53770 и CVE-2025-53771.

Основные рекомендации по предотвращению таких вторжений сводятся к срочному обновлению уязвимых продуктов SharePoint, а также к использованию антивирусов, которые способны детектировать эксплоиты ToolShell.

Однако, как показывает опыт, уязвимые системы зачастую остаются непропатченными ещё долгое время, а модификация кода вредоносов помогает злоумышленникам обойти антивирус.

В данной статье описывается общий подход к детектированию атак на основе уязвимостей десериализации в среде CLR, таких как CVE-2025-53770. В конце статьи вы найдёте YARA-правило, которое позволяет выявлять подозрительные сборки (эксплуатацию).

Что делает ToolShell

Этот эксплоит существует только внутри CLR (среда исполнения .NET) и только благодаря такой фишке, как десериализация. Десериализация нужна, чтобы «оживлять» объекты CLR из байтов: передавать их по сети и между процессами, восстанавливать состояние из файлов, БД, логов и очередей, делать снимки/кэши и реплики для отказоустойчивости и воспроизводимости. Это универсальный мост от протоколов и хранения (REST/gRPC, брокеры сообщений, snapshot'ы) к живой объектной модели: графы с ссылками и версиями восстанавливаются без ручного клея, что упрощает интеграцию, миграцию и долговременное хранение данных.

Десериализация опасна тем, что при чтении потока среда может создать «неожиданные» типы и вызвать их скрытые колбэки (например, ISerializable, [OnDeserialized], IDeserializationCallback) в обход конструкторов — это открывает путь к gadget-цепочкам с побочными эффектами вплоть до RCE/DoS.

Особенно уязвимы механизмы, которые несут имя типа в данных: BinaryFormatter (признан принципиально небезопасным, помечен как устаревший и по умолчанию блокируется), NetDataContractSerializer, ObjectStateFormatter. Просто посмотрите примеры по этой ссылке по поиску «serial».

Эксплойт ToolShell отправляет специально сформированный POST-запрос на SharePoint-страницу /_layouts/15/ToolPane.aspx с параметрами MSOtlPn_Uri и MSOtlPn_DWP, содержащими вредоносный payload. Внутри параметра MSOtlPn_DWP передаётся XML-разметка веб-части, с помощью которой SharePoint создаёт объект типа

Microsoft.PerformancePoint.Scorecards.ExcelDataSet и устанавливает его свойство CompressedDataTable равным Base64-строке с вредоносными данными.

При попытке доступа к свойству DataTable этого объекта происходит десериализация переданных данных: метод GetObjectFromCompressedBase64String декодирует Base64-строку, распаковывает её (данные сжаты) и вызывает функцию BinarySerialization.Deserialize из библиотеки SharePoint для получения объекта из байтового потока.

Главная особенность, на которую хочу обратить внимание: уязвимость приводит к выполнению управляемого .NET-кода, который создаётся динамически в рантайме. То есть вредоносный объект, полученный из десериализации, не существует изначально как загруженная DLL — он формируется «на лету».

CLR, в свою очередь, не может ни выполнить MSIL-код напрямую, ни работать с переданным объектом, поэтому перед выполнением код компилируется JIT-компилятором в машинный код, а переданный объект, собственно, сериализуется. И за это всё среда CLR отвечает сама (программист прикладного приложения не писал этот код).

Таким образом, в момент эксплуатации мы должны будем наблюдать признаки динамической компиляции и загрузки кода в процессе SharePoint (IIS/ASP.NET), ну и, само собой, какую-то активность для десериализации. Это уже некоторая точка для начала детекта (смотреть не на данные, а на саму CLR). Но пока ничего не понятно.

Что происходит при вызове BinaryFormatter.Deserialize

В уязвимом коде внутренний BinarySerialization.Deserialize — это часть ADO.NET, это скрытый за свойством SerializationFormat.Binary, который внутри полагается на BinaryFormatter, поэтому тоже признан небезопасным.

Когда вызывается десериализация с байтовым потоком, CLR выполняет серию шагов по восстановлению объектов из этих данных:

(1) Чтение метаданных сериализации. Сначала сериализатор читает метаданные, чтобы понять, что куда сериализовать. BinaryFormatter считывает заголовок сериализации. Естественно, поток в бинарном формате .NET Remoting (NRBF) содержит записи с именами сборок и типов (BinaryLibrary, ClassInfo, MemberTypeInfo и др.).

Это значит, что где-то будет поток, в котором прямым текстом сказано: «Я мегаопасный тип, и сейчас я буду опасно десериализоваться». Отсюда напрашивается практическая идея: искать в памяти/файлах NRBF-потоки и матчить подозрительные FQTN (типы-гаджеты) простым сигнатурным сканом/YARA по строкам имён сборок/типов.

Спецификация формата и грамматика записей официально опубликованы Microsoft — это не составит труда. Значит, можно перебрать все гаджеты и составить список подозрительных данных, которые, если десериализуются, то могут вызвать RCE. Эти данные должны лежать в памяти приложения, их можно выявить YARA-сканером. Это хорошо.

(2) Загрузка сборки и типа. Для каждой записи объекта форматтер берёт *имя типа + имя сборки* и пытается связать их с System.Type текущего домена загрузки. Поведение контролируют: BinaryFormatter.Binder (кастомная привязка типов) и AssemblyFormat/FormatterAssemblyStyle (как именно искать сборку). По умолчанию — обычное разрешение сборок CLR (через Assembly.Load и стандартные пути/загрузочные

контексты); при нехватке — может сработать AppDomain.AssemblyResolve, если обработчик подписан.

В .NET Framework это включало и GAC; в .NET (Core/5+) — другие правила контекстов/пробинга, но суть та же: если сборка/тип находятся, они будут загружены/разрешены.

Именно поэтому *полиморфная десериализация по данным* небезопасна: поток диктует, какие **конкретные** типы создать. Эти попытки загрузки отлично видны в ETW по провайдеру CLR (Microsoft-Windows-DotNETRuntime) и категории Loader — можно подписаться и видеть AssemblyLoad/ModuleLoad.

(3) Выделение и инициализация объекта. После разрешения типа десериализатор создаёт экземпляр объекта. В нашем случае — злодейский экземпляр объекта, и более того – этот объект реализует опасный тип данных (об опасных типах ниже).

После разрешения типа форматтер *может создать объект без вызова обычных конструкторов* и начать исполнять «магические» колбэки десериализации:

- [Serializable] без ISerializable → аллокация через FormatterServices.GetUninitializedObject (в новых версиях также есть GetSafeUninitializedObject) + прямое заполнение полей; это опасно, потому что инварианты конструкторов обойдены.
- ISerializable/«спец-ctor» ctor(SerializationInfo, StreamingContext) → конструктор выполняется немедленно во время десериализации (код класса уже исполняется).
- IObjectReference → сначала временный объект, затем вызов GetRealObject() подменяет его «настоящим» (часто используемая ступень гаджет-цепочек).
- IDeserializationCallback.OnDeserialization и методы с [OnDeserialized] → вызываются после восстановления графа; тоже произвольный код типа в «горячей» точке.

(4) Сборка графа и подстановки. CLR ведёт ObjectManager: трекает объекты/ссылки/циклы, выполняет фиксации ссылок и под конец поднимает события десериализации. Ошибки на этом этапе — классические индикаторы сломанного/злонамеренного потока.

BinaryFormatter доверяет потоку — и даёт доступ к мощным примитивам жизненного цикла объектов (аллокация без конструкторов, спец-контрукторы, подмена объектов, колбэки), а также к разрешению произвольных типов из доступных сборок. Это делает возможными gadget-цепочки с побочными эффектами вплоть до RCE/DoS.

Поэтому Microsoft официально классифицирует BinaryFormatter как принципиально небезопасный и рекомендует немедленную миграцию. Отличное обзорное подтверждение — официальное руководство по безопасности и статус обесценивания/удаления API. Для глубины техники см. также классический разбор Форшоу (долгих лет и плодотворного творчества, мой герой).

Какие бывают опасные гаджеты

Цель злоумышленника — внедрить гаджет-объекты (gadget objects): экземпляры классов, чьё поведение при десериализации может привести к выполнению кода. Такие типы, как правило, не предназначены для использования в бизнес-логике (там всякие данные, картинки и ничего исполняемого), но доступны в .NET Framework.

Их называют «исполняемыми» типами, поскольку они содержат точки выполнения кода при десериализации (например, в конструкторе сериализации или обратных вызовах).

Что происходит при вредоносной нагрузке?

(1) Загрузка неожиданного типа. В потоке может быть указан тип System.DelegateSerializationHolder, System.Data.DataSet, System.Workflow.ComponentModel.Serialization.ActivitySurrogateSelector и тому подобное.

CLR загрузит соответствующую сборку (например, mscorlib, System.Workflow, System.Data) и тип — даже если приложение не планировало это делать.

(2) Выполнение кода в конструкторе сериализации или обратных вызовах.

- Делегаты**:** Если payload содержит DelegateSerializationHolder+DelegateEntry, CLR создаст DelegateSerializationHolder, а затем вызовет GetRealObject(). Внутри этого метода вызывается Delegate.CreateDelegate, создавая реальный делегат, указывающий на нужный метод — злоумышленник может указать, например, System.Diagnostics.Process.Start. Если сигнатуры не совпадают, CLR сгенерирует динамический stub-метод (adapter), хранящийся в динамической сборке — это позволяет вызвать метод с нужной сигнатурой.
- ActivitySurrogateSelector: Один из мощнейших гаджетов. Он используется для подмены сериализуемого типа.
- Через SerializationInfo.SetType злоумышленник заявляет, что объект — это System.Windows.Forms.AxHost.State, и добавляет ключ "PropertyBagBinary" со вложенным потоком.
- Внутри AxHost.State при десериализации выполняется повторная десериализация этих данных — это и есть следующий этап атаки. Внутри передаётся уже реальная полезная нагрузка: скомпилированная сборка, загружаемая через Assembly.Load(byte[]). В момент её загрузки появляется **динамическая сборка** без PE-образа.
- Другие гаджеты:
    - System.Runtime.Remoting.ObjRef — может создать прокси на удалённый объект, даже подключиться к внешнему серверу.
    - System.Data.DataSet — может выполнить повторную загрузку XML-схем и вызвать уязвимости XML (XXE и др.).
    - System.Windows.Data.ObjectDataProvider — может вызвать произвольный метод при десериализации XAML (в LosFormatter и именно этот гаджет в данном примере, который гуляет in-the-wild, мы и рассмотрим).
    - DesignerVerb, ClaimsPrincipal, TextFormattingRunProperties. — используются в цепочках, где код вызывается, например, при ToString() или в OnDeserialized.

Все эти классы изначально создавались для легитимных нужд — сериализация делегатов, прокси, ActiveX-состояний. Но они включают код, который исполняется при десериализации, что и делает их гаджетами.

В инструментах вроде YSoSerial.NET собраны десятки таких классов-гаджетов. Но найти и перечислить такие паттерны для сканирующего YARA-правила – вполне реально.

Как выглядит злая сборка на SharePoint

А теперь к частностям: давайте посмотрим, что происходит на самом SharePoint. Сначала мы плотно поработаем с таблицами и Excel на нём, снимем дамп и проэксплуатируем, а затем снова снимем дамп.

Было:

Стало:

Добавилось сборок:

Assem	Addr	Flags
App_Web_toolpane.aspx.9c9699a8.pq2eudb2	0x2523f533a40	
Microsoft.GeneratedCode	0x2523f535780	Dynamic
Microsoft.Web.Design.Server.intl	0x2523f5323c0	
PresentationCore	0x2523f534b20	Native
PresentationFramework	0x2523f532720	Native
System.Data.Services	0x2523f536740	
System.Web.Extensions.Design	0x2523f537820	
Microsoft.Web.Design.Server.intl	0x2523f5323c0	
PresentationCore	0x2523f534b20	Native
PresentationFramework	0x2523f532720	Native

*App_Web_toolpane.aspx** — это динамически скомпилированная ASP.NET-сборка страницы toolpane.aspx. Для WebForms ASP.NET компилирует .aspx «на лету» в App_Web_*.dll и подгружает их в домен приложения; сам механизм описан в MSDN («ASP.NET Compilation Overview», «Dynamic Compilation»). Это означает, что соответствующая страница действительно была запрошена/принудительно задействована рантаймом. Для SharePoint это коррелирует с инструментальной страницей «Tool Pane», которая исторически хостится в toolpane.aspx и используется для редактирования/управления веб-частями.

Microsoft.GeneratedCode, Dynamic — это характерное имя **динамической** сборки, которую .NET создаёт, когда XmlSerializer впервые сериализует/десериализует *незнакомый тип*. Генерация идёт через XmlSerializerFactory → XmlSerializer.GenerateTempAssembly(...) → TempAssembly, т.е. происходит именно **эмиссия** вспомогательной сборки и загрузка её в домен.

Механика подтверждается открытыми исходниками .NET Framework: фабрика всегда пытается либо подгрузить предсгенерированную *.XmlSerializers.dll, либо сгенерировать «temp-assembly» в рантайме. **Имя** динамической сборки на практике часто равно Microsoft.GeneratedCode (подтверждается отчётами и практикой вообще), но это не «жёсткий» контракт API, а де-факто поведение конкретной реализации.

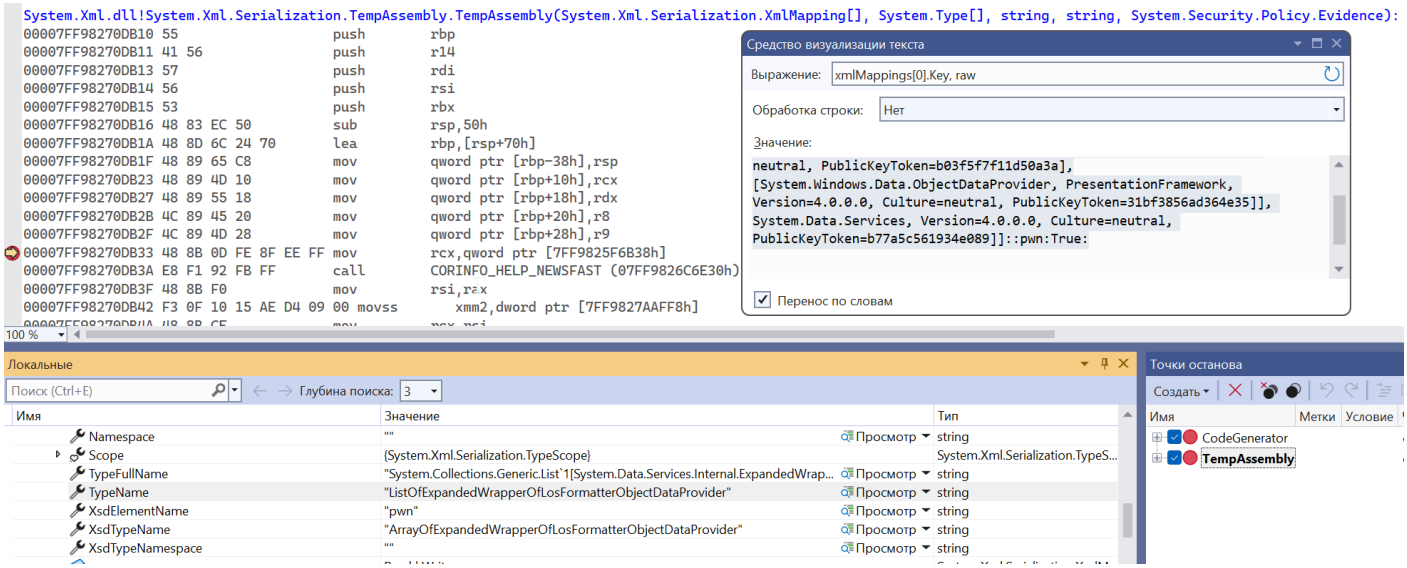
А это именно то, что нам нужно. Найдём именно такую сборку-хелпер — значит, что-то десериализуется.

Давайте изучим сборку: посмотрим на содержимое DumpAfter.txt (там вывод отладчика с командами).

Видны автосгенерированные поля вида *id7_ObjectDataProvider, id19_ExpandedWrapperOfLosFormatter, id20_LosFormatter* — это прямо указывает, что в сформированном графе встречались System.Windows.Data.ObjectDataProvider (WPF/XAML), System.Data.Services.Internal.ExpandedWrapper (WCF Data Services/OData), а также System.Web.UI.LosFormatter (ASP.NET ViewState). Все три — известные опорные точки в десериализационных цепочках.

Сами API задокументированы в Microsoft Docs: ObjectDataProvider (WPF, XAML-порождение объектов), ExpandedWrapper<> (внутренние обёртки eager-loading в ADO.NET Data Services), LosFormatter (и он признан небезопасным, это зафиксировано прямо в документации).

Также давайте подключимся отладчиком и поставим брейкпоинт на **System.Xml.Serialization.TempAssembly.TempAssembly(System.Xml.Serialization.XmlMapping[], System.Type[], string, string, System.Security.Policy.Evidence)** — это то место, где рантайм уже собрал отражённые маппинги для типа, под который генерируется сериализатор.



Что мы можем здесь увидеть (debug.txt)?

TypeFullName = "System.Collections.Generic.List`1[System.Data.Services.Internal.ExpandedWrapper`2[System.Web.UI.LosFormatter,System.Windows.Data.ObjectDataProvider]]"

Это конструированный дженерик List<T>, где T = ExpandedWrapper<LosFormatter, ObjectDataProvider>`

Также видим:

- *TypeName = "ListOfExpandedWrapperOfLosFormatterObjectDataProvider"*
- *DefaultElementName = “ArrayOfExpandedWrapperOfLosFormatterObjectDataProvider”*
- *Name = "ObjectDataProvider"*
- *Name = "ProjectedProperty0"*
- *Name = "ExpandedWrapperOfLosFormatterObjectDataProvider"*

Это работа внутренних помощников CodeIdentifiers, которые «выпрямляют» имена типов (снимают «галочки» из объектов вроде “ExpandedWrapper`2” и склеивают “Of”)

А ещё можно увидеть:

Key "System.Collections.Generic.List`1[[System.Data.Services.Internal.ExpandedWrapper`2[[System.Web.UI.LosFormatter, System.Web, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a],[System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]], System.Data.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]]::pwn:True:"

Тут строкой зашита assembly-qualified name конструированного типа с вложенными AQN аргументов дженерика и XML-элементом **pwn** (корневое имя злодейского параметра через XmlRoot).

Это та самая строка, по которой удобно делать сигнатурный поиск/YARA — в ней есть РЕШИТЕЛЬНО ВСЁ.

Как бы то ни было, данные ещё не десериализировались, но в памяти служебной сборки десериализации появилось невероятное количество «подозрительных» артефактов. Да собственно, одной строкой TypeName/Key видно, что сериализатор должен сгенерировать код для List<ExpandedWrapper<LosFormatter,ObjectDataProvider>>, а ниже — что внутри ObjectDataProvider сериализуются поля, позволяющие вызвать произвольный метод с параметрами. Это точно не должно встречаться при «обычной» работе SharePoint.

И в целом, исследование вспомогательных сборок для сериализации — это невероятное поле для того, чтобы собрать артефактов для детектирования самого факта, что пришёл пэйлоад, который рантайм собирается десериализовать в что-то явно очень нехорошее.

Также, если мониторить, что вызывается в пространствах CodeGenerator, то мы заметим конструктор методов (в нашем случае 4 метода), который будет строить методы, и мы можем попробовать взять m_ILStream , когда динамический метод будет готов.

Я не привожу листинг отладчика, но в YARA-правило добавил и этот артефакт — он будет повторяем (это ещё не NGEN метод, поэтому он будет одинаков при конструировании методов, отвечающих за сериализацию одинаковых типов). Я взял значительную часть потока IL, которая появилась для выравнивания данных. Эмпирически — это находит злые десериализаторы, и не встречается в хороших. Но это уже даже несколько избыточно.

Описанные проверки мы и будем использовать в YARA-правиле, позволяющем найти сборку-помощника, которая используется для злой десериализации.

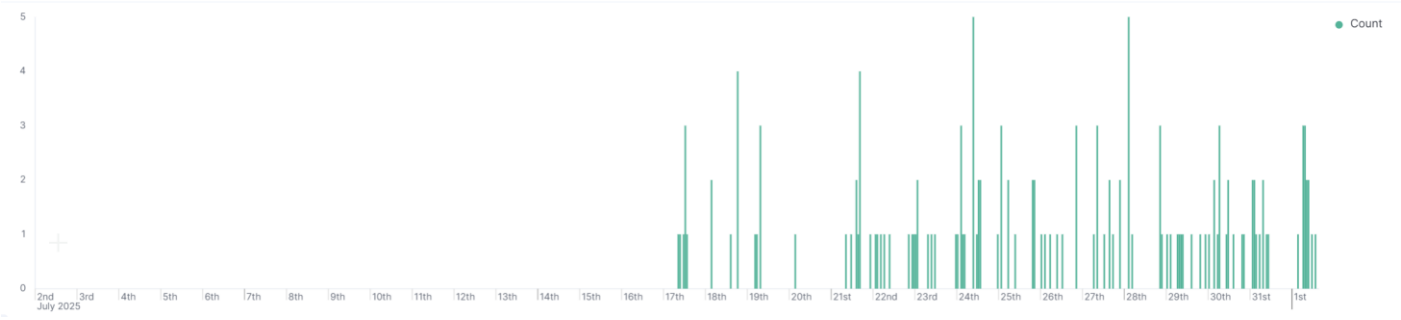
Сам вредоносный код загружается следующим этапом.

Общий порядок детектирования

Теперь опишем целостно, как может работать система детектирования эксплойта на десериализацию:

1. ETW-монитор EDR-решения запускается при старте системы (либо постоянно слушает). Он подписан на события *CLR: Loader.*
2. При получении события *Loader* (логическое *И*):
    a. Динамический *Microsoft.GeneratedCode* без пути
    b. *App_Web_toolpane.aspx** без пути
3. Проверили Yara-правилом, как показано выше, что *Microsoft.GeneratedCode* — это хелпер, который десериализует гаджет для уязвимого параметра. Очень сильно насторожились, если всё так. *И* возможно, даже отфорензили компьютер.
4. Сканируем память того же домена приложений на паттерны известных гаджетов. Если найдено, то это уже 100% эксплуатация.

На момент написания этой статьи в мониторинге нашего MDR за последние 30 дней было всего 187132 события, когда IIS (который хостит всё, что угодно) вызвал загрузку *Microsoft.GeneratedCode*. Ну да, службы IIS занимаются десериализацией. Однако, вот как выглядит за те же 30 дней статистика загрузки скомпилированного ASP *App_Web_toolpane* в домен приложения, на котором работает Sharepoint:



Таких событий мало – это хороший пункт для алерта. А поиск «правильного» хелпера и сканирование на наличие опасных гаджетов позволяет совсем исключить ложные срабатывания.

Вот пример YARA-правила, в котором используются вышеупомянутые строки из DumpAfter.txt и данные из m_ILStream:

```
rule ToolShell_MicrosoftGeneratedCode_XmlGadget
{
    meta:
        description = "XmlSerializer helper assembly loaded during ToolShell exploit. Based on
pwn object deserialization"
        author = "Rodchenko"
        reference = "CVE-2025-53770 / ExpandedWrapper + LosFormatter chain"
```

```
        hash_mode = "memory "

    strings:
        /* IL-сигнатуры, которые взяли из m_ILStream когда
           динамический метод "Microsoft.GeneratedCode" только строился
           https://stackoverflow.com/a/4147132/6300544
        */
        $il4 = { 17 3B 00 00 00 00 05 28 0C 00 00 0A 0A 06 D0 0A 00 00 01 28 0D 00 00 0A FE 01
39 00 00 00 00 38 00 00 00 00 02 05 28 0E 00 00 0A 7A 02 03 04 05 16 14 28 0F 00 00 0A 0E 05 }

        $data =
"System.Collections.Generic.List`1[[System.Data.Services.Internal.ExpandedWrapper`2[[System.Web
.UI.LosFormatter, System.Web, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b03f5f7f11d50a3a],[System.Windows.Data.ObjectDataProvider,
PresentationFramework, Version=4.0.0.0, Culture=neutral, PublicKeyToken=31bf3856ad364e35]],
System.Data.Services, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089]]"
        $TypeName = "ListOfExpandedWrapperOfLosFormatterObjectDataProvider"

        $idpwn = "id1_pwn"
        /*
        $id2 = "id2_Item"
        $id3 = "id3_Item"
        $id4 = "id4_Description"
        $id5 = "id5_ExpandedElement"
        */
        $id6 = "id6_ProjectedProperty0"
        $id7 = "id7_ObjectDataProvider"
        /*
        $id8 = "id8_IsInitialLoadEnabled"
        $id9 = "id9_ObjectType"
        $id10 = "id10_ObjectInstance"
        $id11 = "id11_MethodName"
        $id12 = "id12_ConstructorParameters"
        */
        $id13 = "id13_anyType"
        /*
        $id14 = "id14_MethodParameters"
        $id15 = "id15_IsAsynchronous"
        $id16 = "id16_MemberInfo"
        $id17 = "id17_Type"
        $id18 = "id18_DataSourceProvider"
        */
        $id19 = "id19_ExpandedWrapperOfLosFormatter"
        $id20 = "id20_LosFormatter"
        /*
        $id21 = "id21_Item"
        */
    condition:
        $il4 or $data or $TypeName or $id19 or $id7 or $id6 or ($idpwn and 3 of ($id*))
}
```