SSO Wars: The Token Menace

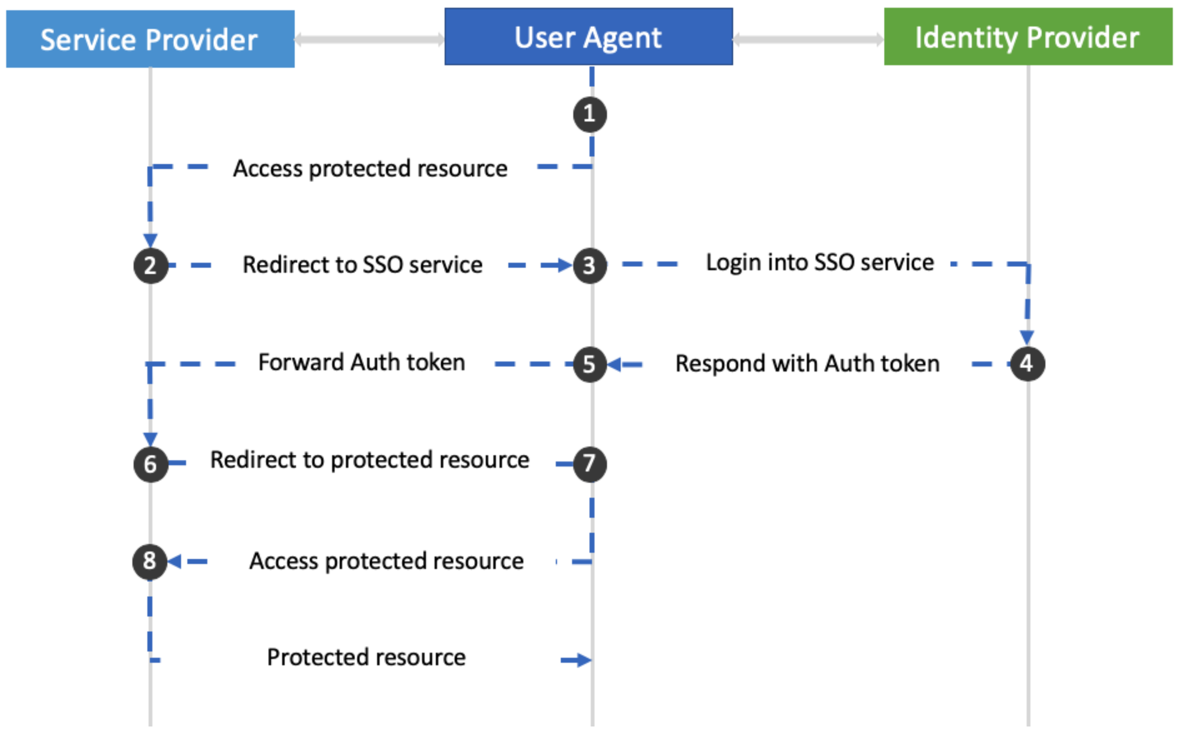
Oleksandr Mirosh & Alvaro Muñoz

Table of Contents

Introduction

Authentication and authorization are critical security controls of any application that operates on sensitive data. Any vulnerability that allows attackers to subvert these controls can open the door to your applications and grant attackers access to sensitive application data. We reviewed how delegated authentication is enforced in .NET frameworks and found several vulnerabilities ranging from authentication bypass, to denial of service, and even arbitrary code execution.

Before diving into the discovered bugs and flaws, we will briefly introduce the concept of delegated authentication, also referred to as federated identities or single sign on (SSO) [1]. The basic concept is that an application or service provider will delegate the authentication of its users to a third-party service or identity provider.



When an unauthenticated user tries to access the service provider (1), it is redirected to the Single-Sign-On service (2, 3), which asks for its credentials and proceeds to validate them. If the presented credentials are valid, an authentication token is emitted (4) and delivered to the user (5).

Authentication tokens can be presented in different forms, but in general they should contain at least the following attributes:

- Issuer of the token (indicates who issued the token)
- Audience of the token (indicates who is meant to use the token)
- Usage conditions that can include:
  - Restriction of usage before a date
  - Restriction of usage on or after a date
- Identity of the authenticated user

- Known attributes or claims for the authenticated user that can include:
    - Email address
    - User ID
    - Roles
- Authentication statements that can include:
    - Authentication mechanism
    - Authentication instant
- Signature

From a security standpoint, the signature is crucial because it can guarantee that nobody other than the issuer modified the token attributes.

The user presents this token to the service provider (6), which needs to validate it before allowing the user access. As part of this validation, the service provider must verify that:

- The token has not been tampered with
- The issuer of the token is trusted
- The token is meant for the service provider
- General conditions such as token lifetime and replay checks are met.

If the validation is successful, the service provider grants the user access to the resource (7, 8).

This protocol is secure in design, however specific implementations might introduce vulnerabilities that can subvert the security of the whole authentication process. If we put our black hat on for a moment, we can think about how to attack this scenario. For example, an attacker could think of attacking the identity provider in order to try to inject additional claims in the issued token or even try to gain access to the keys used for signing these tokens.
We, however, focused on different vectors, specifically:

- Token parsing vulnerabilities (normally before or during signature verification)
- Signature verification vulnerabilities

We found vulnerabilities in the .NET libraries reviewed for each of the above vectors. In the following sections, we review and explain the details of these vulnerabilities.

Arbitrary Constructor Invocation

CVE-2019-1083 [2]

This vulnerability is an example of the “Token parsing vulnerabilities” mentioned in the introduction. Even if the signature verification is 100% secure, an application still reads and parses the authentication tokens and uses certain token fields for signature verification. Such fields are very attractive to an attacker because they are used before the application knows whether the signature is valid or not. We reviewed the way that .NET libraries parse two authentication token formats: JSON Web Token (JWT) [3] and SAML [4] and found that an attacker can provide an arbitrary value for the signature algorithm string, which the system uses to instantiate an arbitrary, user-controlled Type. In this section, we review the details of the vulnerability and the potential effects.

JWT is an Internet standard for creating JSON-based access tokens that asserts some number of claims. JWT relies on other JSON-based standards: JSON Web Signature [5] and JSON Web Encryption [6]. Both standards have header fields that identify which algorithm is used to generate the signature (in the case of JSON Web Signature) and to decrypt the message (in the case of JSON Web Encryption). Header fields are processed before the signature verification. We noticed that some JWT implementations in .NET have a security weakness when processing the "`alg`" field of a JWT Header. For example, this is how `System.IdentityModel.Tokens.Jwt` library [7], which is used by Microsoft Exchange Server, processes this value:

First, notice that there is no restriction or validation of the string value from the "`alg`" field:

```
// System.IdentityModel.Tokens.JwtSecurityTokenHandler. Code ref: 1
protected virtual JwtSecurityToken ValidateSignature(string token, TokenValidationParameters
validationParameters)
{
    ...
    JwtSecurityToken jwtSecurityToken = this.ReadToken(token) as JwtSecurityToken;
    ...
    {
        string text = jwtSecurityToken.Header.Alg;
        ...
          try
            {
                if (this.ValidateSignature(bytes, array, securityKey, text))
            …
```

The “`alg`” value string is passed for signature verification:

```
// System.IdentityModel.Tokens.JwtSecurityTokenHandler. Code ref: 1
private bool ValidateSignature(byte[] encodedBytes, byte[] signature, SecurityKey key, string
algorithm)
{
    SignatureProvider signatureProvider = this.SignatureProviderFactory.CreateForVerifying(key,
algorithm);
    ...
```

```
// System.IdentityModel.Tokens.SignatureProviderFactory. Code ref: 2
public virtual SignatureProvider CreateForVerifying(SecurityKey key, string algorithm)
{
    return SignatureProviderFactory.CreateProvider(key, algorithm, false);
}
```

```
// System.IdentityModel.Tokens.SignatureProviderFactory. Code ref: 2
private static SignatureProvider CreateProvider(SecurityKey key, string algorithm, bool
willCreateSignatures)
{
    ...
    AsymmetricSecurityKey asymmetricSecurityKey = key as AsymmetricSecurityKey;
    if (asymmetricSecurityKey != null)
    {
        ...
        return new AsymmetricSignatureProvider(asymmetricSecurityKey, algorithm,
willCreateSignatures);
    }
    ...
}
```

There is similar code to process `SymmetricKey` by using `SymmetricSignatureProvider` instead. Both providers are similar and invoke `GetHashAlgorithmForSignature`, therefore, we will focus on `AsymmetricSignatureProvider`:

```
// System.IdentityModel.Tokens.AsymmetricSignatureProvider. Code ref: 3
public AsymmetricSignatureProvider(AsymmetricSecurityKey key, string algorithm, bool
willCreateSignatures = false)
{
    ...
        this.hash = this.key.GetHashAlgorithmForSignature(algorithm);
    ...
}
```

Both `System.IdentityModel.Tokens.RsaSecurityKey` and `System.IdentityModel.Tokens.X509AsummetricSecurityKey` have the next line of code:

```
// System.IdentityModel.Tokens.X509AsymmetricSecurityKey. Code ref: 4
public override HashAlgorithm GetHashAlgorithmForSignature(string algorithm)
{
    ...
    object algorithmFromConfig = CryptoHelper.GetAlgorithmFromConfig(algorithm);
    ...
```

And our arbitrary string is passed to `System.Security.Cryptography.CryptoConfig.CreateFromName():`

```
// System.IdentityModel.CryptoHelper.Code ref: 5
internal static object GetAlgorithmFromConfig(string algorithm)
{
    ...
            try
            {
                obj = CryptoConfig.CreateFromName(algorithm);
            }
```

This method allows for the instantiation of any arbitrary public Type that has a public no-arguments constructor:

```
// System.Security.Cryptography.CryptoConfig. Code ref: 6
public static object CreateFromName(string name, params object[] args)
{
    ...
    if (type == null) {
            type = Type.GetType(name, false, false);
            if (type != null && !type.IsVisible) type = null;
    }
    ...
    RuntimeType runtimeType = type as RuntimeType;
    if (runtimeType == null) return null;
    if (args == null) args = new object[0];
    MethodBase[] array = runtimeType.GetConstructors(BindingFlags.Instance |
BindingFlags.Public | BindingFlags.CreateInstance);
    if (array == null) return null;
    List<MethodBase> list = new List<MethodBase>();
    for (int i = 0; i < array.Length; i++){
            MethodBase methodBase = array[i];
            if (methodBase.GetParameters().Length == args.Length){
            list.Add(methodBase);
            }
    }
    if (list.Count == 0) return null;
    array = list.ToArray();
    object obj;
    RuntimeConstructorInfo runtimeConstructorInfo =
Type.DefaultBinder.BindToMethod(BindingFlags.Instance | BindingFlags.Public |
BindingFlags.CreateInstance, array, ref args, null, null, null, out obj) as
RuntimeConstructorInfo;
         ...
    object result = runtimeConstructorInfo.Invoke(BindingFlags.Instance | BindingFlags.Public |
BindingFlags.CreateInstance, Type.DefaultBinder, args, null);
    if (obj != null) Type.DefaultBinder.ReorderArgumentArray(ref args, obj);
    return result;
}
```

Being able to invoke an arbitrary parameter-less constructor does not sound too dangerous, but we describe possible ways to abuse this code later.

This problem not only affects JWT but also other technologies such as SAML Tokens. More details about this type of access token are provided in the next section where we show how to bypass the signature verification. For now, we are interested in only one attribute - `SignatureMethod` of the `SignedInfo` element. Similar to the "`alg`" string value in JWT, the `SignatureMethod` string value is not validated either and is passed to the same crypto methods of the .NET library:

```
 // System.IdentityModel.Tokens.SamlAssertion. Code ref: 7
private void VerifySignature(SignedXml signature, SecurityKey signatureVerificationKey)
{
    ...
    signature.StartSignatureVerification(signatureVerificationKey);
    ...
}
```

```
// System.IdentityModel.SignedXml. Code ref: 8
public void StartSignatureVerification(SecurityKey verificationKey)
{
    string signatureMethod = this.Signature.SignedInfo.SignatureMethod;
    ...
    AsymmetricSecurityKey asymmetricSecurityKey = verificationKey as AsymmetricSecurityKey;
    ...
    using (HashAlgorithm hashAlgorithmForSignature =
asymmetricSecurityKey.GetHashAlgorithmForSignature(signatureMethod))
```

Any .NET application that uses SAML tokens, along with some that use JWT, allows an unauthenticated attacker to load any .NET library from Global Assembly Cache (GAC), or the working directory, and instantiate any arbitrary public Type that has a public no-argument constructor.

As we mentioned before, at first glance, this does not seem to be a serious security problem because an attacker cannot control any data. However, there are many available libraries and Types which are worth reviewing. Additionally, the assumption that an attacker is not able to control any data might not always be the case. For example, an attacker can look for gadgets that read request parameters, cookies, or headers because the .NET request object is retrieved from a static method. Let's take a look at two constructors from the .NET Framework:

```
// System.Web.Mobile.CookielessData. Code ref: 9
public CookielessData()
{
    string formsCookieName = FormsAuthentication.FormsCookieName;
    string text = HttpContext.Current.Request.QueryString[formsCookieName];
    ...
    {
        FormsAuthenticationTicket tOld = FormsAuthentication.Decrypt(text);
        ...
```

and:

```
// System.Web.Security.PassportIdentity. Code ref: 10
public PassportIdentity()
{
    HttpContext current = HttpContext.Current;
    ...
        string szQueryStrT = current.Request.QueryString["t"];
        string szQueryStrP = current.Request.QueryString["p"];
        HttpCookie httpCookie = current.Request.Cookies["MSPAuth"];
        HttpCookie httpCookie2 = current.Request.Cookies["MSPProf"];
        HttpCookie httpCookie3 = current.Request.Cookies["MSPProfC"];
        string text = (httpCookie != null && httpCookie.Value != null) ? httpCookie.Value : 
string.Empty;
        string text2 = (httpCookie2 != null && httpCookie2.Value != null) ? httpCookie2.Value : 
string.Empty;
        string text3 = (httpCookie3 != null && httpCookie3.Value != null) ? httpCookie3.Value : 
string.Empty;
        StringBuilder stringBuilder = new StringBuilder(1028);
        StringBuilder stringBuilder2 = new StringBuilder(1028);
        text = HttpUtility.UrlDecode(text);
        text2 = HttpUtility.UrlDecode(text2);
        text3 = HttpUtility.UrlDecode(text3);
        int errorCode = UnsafeNativeMethods.PassportCreate(szQueryStrT, szQueryStrP, text, 
text2, text3, stringBuilder, stringBuilder2, 1024, ref this._iPassport);
```

In both cases, an attacker can control data used in such constructors. As we saw in the examples above, potential attacks depend on the list of available Types on the target systems, in a similar way to Unsafe Deserialization attack "gadgets".

We decided to review two of the most important Microsoft servers: Microsoft Exchange 2019 and Microsoft SharePoint 2019 Server since this vulnerability affected both of them:

- Microsoft Exchange 2019: vulnerable through JWT Token and SAML parsers
- Microsoft SharePoint 2019: vulnerable code is reachable from SAML parsers

We found examples of constructors that use data read from `HttpContext.Current.Request` in both products, but were not able to get any valuable results from them. However, other Types turned out to have interesting behaviors which may enable an attacker to perform any of the following attacks.

Potential Attack Vectors

Information Leakage

For example, SharePoint server returns different results whether Type resolution and instantiation was successful or not. This potentially enables an attacker to collect information about available libraries and products on the target server.

Denial of Service

Along with the instance constructors, it is also possible to execute code in the static constructors. Many static constructors perform initialization that may be useful for DoS attacks or could change some configurations of the application that can enable part of a more complicated attack when chained with other vulnerabilities.

A good example of gadgets that lead to a Denial of Service are described next:

```
// System.Management.Automation.Remoting.WSManPluginManagedEntryInstanceWrapper. Code ref: 11
```

```
~WSManPluginManagedEntryInstanceWrapper()
{
    this.Dispose(false);
}
```

In this case, the triggering code is in the `Finalize()` method rather than the constructor.

```
// System.Management.Automation.Remoting.WSManPluginManagedEntryInstanceWrapper. Code ref: 11
private void Dispose(bool disposing)
{
    if (this.disposed)  return;
    this.initDelegateHandle.Free();
    this.disposed = true;
}
```

The handle `initDelegateHandle` is never initialized.

```
// System.Runtime.InteropServices.GCHandle. Code ref: 12
public void Free()
{
    IntPtr handle = this.m_handle;
    if (handle != IntPtr.Zero && Interlocked.CompareExchange(ref this.m_handle, IntPtr.Zero,
handle) == handle) {
        ...
        } else {
        throw new
InvalidOperationException(Environment.GetResourceString("InvalidOperation_HandleIsNotInitialize
d"));
        }
}
```

Because it can raise an unhandled exception, it allowed us to make SharePoint unavailable for processing new requests for about two minutes resulting in a Denial of Service.

Arbitrary Code Execution

A different piece of data controlled by the attacker is the name of the Type to be resolved and instantiated. An attacker can supply the assembly-qualified name, which includes the name of the assembly from which we want to load the Type [6].

.NET provides a way for developers to control assembly resolving [8]. Such custom assembly resolvers may be vulnerable to various attacks. We already mentioned that a lot of static constructors perform initialization procedures such as installing custom assembly resolvers. As a result, we can think of the following two-steps attack vector:

1. Sending name of a Type which instantiation invokes a static constructor that installs a vulnerable assembly resolver and
2. Sending type name with an assembly name that the system won't be able to resolve and therefore will delegate its resolution to installed assembly resolvers. The vulnerable assembly resolver installed in 1) will handle our assembly name triggering the vulnerability.

An example of a types with a static constructor installing a vulnerable assembly resolver in Exchange Server Types is:

```
// Microsoft.Exchange.Search.Fast.FastManagementClient
```

```
static FastManagementClient()
{
    ...
    AppDomain.CurrentDomain.AssemblyResolve += new
ResolveEventHandler(FastManagementClient.OnAssemblyResolveEvent);
}
```

The static constructor for `FastManagementClient` installs a custom assembly resolver with the following code:

```
// Microsoft.Exchange.Search.Fast.FastManagementClient
private static Assembly OnAssemblyResolveEvent(object sender, ResolveEventArgs args)
{
    string str = args.Name.Split(new char[]{','})[0];
    string text = Path.Combine(FastManagementClient.fsisInstallPath, "Installer\\Bin");
    string text2 = Path.Combine(FastManagementClient.fsisInstallPath, "HostController");
    string[] array = new string[] {text,text2};
    for (int i = 0; i < array.Length; i++)
    {
        string text3 = array[i] + Path.DirectorySeparatorChar.ToString() + str + ".dll";
        if (File.Exists(text3))
        {
            return Assembly.LoadFrom(text3);
        ...
```

The Type name controlled by the attacker is available in `ResolveEventArgs.Name`. We can use "`\..\`" in assembly name to change the directory to any location on the current disk, giving us the ability to load any .dll file from any arbitrary folder, and after that instantiate an arbitrary Type from it.

Such an attack may depend on the configuration of the target server. For example `AppDomain.CurrentDomain.AssemblyResolve` may already have some assembly resolvers installed and they are called before the one we installed. For a successful attack, these resolvers should neither resolve the assembly, nor raise an exception. This may be the case when we use "`\..\`" as part of our assembly name because normally assembly resolvers have the following code (the `AssemblyName` constructor raises an exception if there are such characters in the name):

```
AssemblyName assemblyName = new AssemblyName(args.Name);
```

Another example of such gadgets in Exchange Server is:

```
// Microsoft.Forefront.Filtering.Management.PowerShell.Task
static Task()
{
    AppDomain.CurrentDomain.AssemblyResolve += new
ResolveEventHandler(Task.AssemblyResolveEventHandler);
```

We can invoke it by instantiating any child Types (for example, `Microsoft.Forefront.Filtering.Management.PowerShell.Commands.StartDiag`) assembly resolver looks like:

```
// Microsoft.Forefront.Filtering.Management.PowerShell.Task
private static Assembly AssemblyResolveEventHandler(object sender, ResolveEventArgs args)
{
    string binPath = ProductInfo.Instance.BinPath;
    if ((!args.Name.StartsWith("Microsoft.Forefront.", StringComparison.OrdinalIgnoreCase) &&
!args.Name.StartsWith("DiagHelper", StringComparison.OrdinalIgnoreCase)) ||
string.IsNullOrEmpty(binPath))
    {
        return null;
    }
    Assembly result;
    try
    {
        DirectoryInfo arg_6A_0 = new DirectoryInfo(binPath);
        int num = args.Name.IndexOf(',');
        string fileName = args.Name;
       if (num != -1)
        {
            fileName = args.Name.Substring(0, num);
        }
        string text = Task.FindAssembly(arg_6A_0, fileName);
        if (text != null)
        {
            Assembly assembly = Assembly.LoadFrom(text);
            ...
```

Let's review `Task.FindAssembly` code:

```
// Microsoft.Forefront.Filtering.Management.PowerShell.Task
private static string FindAssembly(DirectoryInfo dir, string fileName)
{
    string result;
    try
    {
        if (File.Exists(Path.Combine(dir.FullName, fileName + ".dll")))
        {
            result = Path.Combine(dir.FullName, fileName + ".dll");
        }
        else if (File.Exists(Path.Combine(dir.FullName, fileName + ".exe")))
        {
            result = Path.Combine(dir.FullName, fileName + ".exe");
        }
        ...
    return result;
}
```

This gadget is similar to the previous one, but it also enables us to instantiate Types from `*.exe` files.

Putting it all together, as unauthenticated users, we are able to execute a public no-argument constructor of any public Type from .NET `dll` or `exe` files (from any arbitrary folder on disk). Of course, there is a remaining open question: how can an unauthenticated user upload malicious dll or exe files to the server? We think it can be a very tough task, but it can be significantly easy if an attacker has an account on the target server.

Dupe Key Confusion (XML Signature verification bypass)

CVE-2019-1006 [9]

This attack is an example of the "signature verification vulnerabilities" mentioned in the introduction. Microsoft uses federated identities to allow users to single sign-on on one server and get a token that grants them access to service providers that trust the token issuer. For that purpose, Microsoft leverages multiple protocols, but the ones that are used the most are WS-Federation [10] and SAMLP [11]. It is not the purpose of this paper to explain how these protocols work, the only thing that is relevant for this research is that both of them use SAML tokens to hold the authentication and authorization claims exchanged between the identity provider (IdP) and the service provider (SP) [12].

SAML tokens use XML signatures to make sure these claims are not tampered with by anyone able to intercept the traffic or forge them from scratch. The model is very simple, the IdP should sign the SAML assertion with its private key and SP should verify the signature using IdP's public key. Because an attacker could potentially re-sign the SAML assertion with its own private key and then send the public key for verification, a second critical step is to validate the trust on the signing party by validating the issuer of the public key used for signing the assertion.

XML Signature [13] is used to sign an XML element and then provide the signature value within <SignatureValue/> and public key to be used for signature verification in the <KeyInfo/> tag.

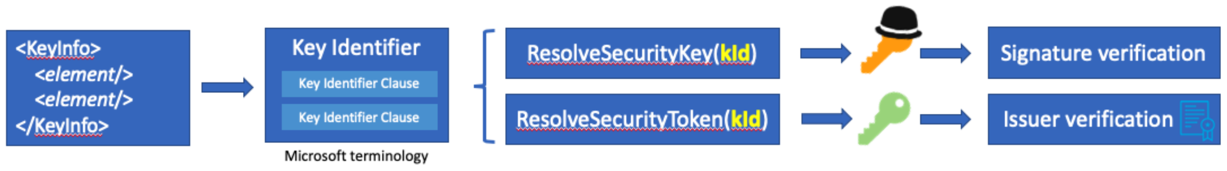
1. XML Signature Validation involves the following operations:
    a. Digest Validation: Digest is verified by retrieving the referenced resource, applying specified transformations and then applying the specified digest method to it. The result is compared to the <DigestValue/> stated in the XML signature; if they do not match, validation fails.
    Note: Most of previously disclosed SAML vulnerabilities and attacks targeted this step. Good examples of such attacks are XML Signature Wrapping Attacks [14] or attacks on XML canonicalization [15]. Using these techniques, an attacker is able to add or modify important values without changing the original signature. For example, with XML Signature Wrapping Attacks it is possible to add new XML nodes [16], and with XML canonicalization attacks, an attacker can add XML comments that do not affect signature calculation but change the meaning of token attribute values.
    b. Signature Validation: The key data or identifier is retrieved from the <KeyInfo/> section and the signature is verified using the method specified in <SignatureMethod/>
2. Authentication of signing party: Security token representing the signing party is retrieved from the <KeyInfo/> section and used to verify that the SAML token was signed by a trusted party.

.NET libraries perform this process as follows:
1. Resolve the key from <KeyInfo /> or load it from local store based on the key identifier specified in the <KeyInfo/> tag.
2. Verify the signature.
3. If the signature is valid,
    1. Resolve the security token using same <KeyInfo/> as the one used in 1.
    2. Resolve the key's Issuer of the security token resolved in 3.1)
    3. Verify trust on key's Issuer
    4. If trusted,
        1. Extract Identity and claims.

The main Types responsible for this process in .NET frameworks are `System.IdentityModel.Tokens.SamlSerializer`, `System.IdentityModel.Tokens.SamlSecurityTokenHandler` and `System.IdentityModel.Tokens.Saml2SecurityTokenHandler`.

Similar types also exist in `Microsoft.IdentityModel.Tokens` namespace: `Saml11.Saml11SecurityTokenHandler` and `Saml2.Saml2SecurityTokenHandler`, however, there are no major differences between these two namespaces, and all of them are vulnerable to the attacks described below.

As indicated previously, the same <KeyInfo/> section is processed twice with different purposes: extract the key for signature verification and to extract the issuer of the key (the signing party).



Usually <KeyInfo/> is quite simple and contains a single key identifier that enables the receiving party to look up both, the key and the token required for both verifications. For example, given the following <KeyInfo/> section:

```
<ds:KeyInfo>
    <o:SecurityTokenReference xmlns:o="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd">
        <o:KeyIdentifier ValueType="http://docs.oasis-open.org/wss/oasis-wss-soap-message-
security-1.1#ThumbprintSHA1">aBch5vEWfaLbuuZUP1jalzk3n2s=</o:KeyIdentifier>
    </o:SecurityTokenReference>
</ds:KeyInfo>
```

The sever looks up an x509 certificate that matches the thumbprint provided (`aBch5vEWfaLbuuZUP1jalzk3n2s=`). Public key from this certificate is used for signature verification, and certificate itself is used for authentication of signing party.

At first glance, this process may seem to be safe, but it is not. First, notice that the x509 certificate is looked up from the <KeyInfo/> twice because both verifications require different types of entities (signature verification works with *SecurityKeys* and authentication of signing parties requires Security*Tokens)*, and they use a different code for retrieving required entities from the same <KeyInfo/> element: `SecurityTokenResolver.TryResolveSecurityKey()` and `SecurityTokenResolver.TryResolveToken().`

From a security standpoint, the main question here is the following: is it possible to provide a single <KeyInfo/> that produces disconnected Security Key and Security Token in these two steps?

In general, the answer depends on the implementation of the mentioned methods and configuration of ***SecurityTokenResolver*** itself, but in all cases that we checked, we were able to craft a <KeyInfo/> element in such a way that we were able to produce two different results for these two validation processes. This enables an attacker to perform an authentication bypass by first signing the SAML token using its own key and then getting the target server to authenticate the signing party in the second stage.

In the following sections, we review the token handlers that we analyzed and share the details on how they work and if and how they are vulnerable.

Windows Communication Foundation (WCF)

WCF [17] is a replacement of all earlier Microsoft web service technologies and supersedes ASP.NET web services. Although WCF is based on SOAP, it is an extension of the ASP.NET web service (ASMX) and supports various protocols like HTTP, HTTPS, TCP, Named Pipes, MSMQ, and others.

WCF has two major modes for providing security (Transport and Message), and a third mode -- `TransportWithMessageCredential` -- that combines the two. Message security uses the WS-Security specification to secure messages, we decided to review the Message security implementation because it uses the WS-Security specification to secure messages, it often supports SAML authentication, and it widely uses XML signature for message level protection.

WS-Security uses `System.IdentityModel.Tokens.SamlSerializer.ReadToken()` to read SAML tokens. The code flow from the `ReadToken()` to `ReadSignature()` method is:

```
// System.IdentityModel.Tokens.SamlSerializer. Code ref: 13
public virtual SamlSecurityToken ReadToken(XmlReader reader, SecurityTokenSerializer
keyInfoSerializer, SecurityTokenResolver outOfBandTokenResolver)
{
    if (reader == null)
    {
        throw DiagnosticUtility.ExceptionUtility.ThrowHelperArgumentNull("reader");
    }
    XmlDictionaryReader reader2 = XmlDictionaryReader.CreateDictionaryReader(reader);
    WrappedReader reader3 = new WrappedReader(reader2);
    SamlAssertion samlAssertion = this.LoadAssertion(reader3, keyInfoSerializer,
outOfBandTokenResolver);
    if (samlAssertion == null)
    {
        throw DiagnosticUtility.ExceptionUtility.ThrowHelperError(new
SecurityTokenException(SR.GetString("SAMLUnableToLoadAssertion")));
    }
    return new SamlSecurityToken(samlAssertion);
}
```

```
// System.IdentityModel.Tokens.SamlSerializer. Code ref: 13
public virtual SamlAssertion LoadAssertion(XmlDictionaryReader reader, SecurityTokenSerializer
keyInfoSerializer, SecurityTokenResolver outOfBandTokenResolver)
{
    if (reader == null)
    {
        throw DiagnosticUtility.ExceptionUtility.ThrowHelperArgumentNull("reader");
    }
    SamlAssertion samlAssertion = new SamlAssertion();
    samlAssertion.ReadXml(reader, this, keyInfoSerializer, outOfBandTokenResolver);
    return samlAssertion;
}
```

```
// System.IdentityModel.Tokens.SamlAssertion. Code ref: 14
public virtual void ReadXml(XmlDictionaryReader reader, SamlSerializer samlSerializer,
SecurityTokenSerializer keyInfoSerializer, SecurityTokenResolver outOfBandTokenResolver)
{
    ...
    if
(wrappedReader.IsStartElement(samlSerializer.DictionaryManager.XmlSignatureDictionary.Signature
, samlSerializer.DictionaryManager.XmlSignatureDictionary.Namespace))
    {
        this.ReadSignature(wrappedReader, keyInfoSerializer, outOfBandTokenResolver,
samlSerializer);
```

```
    }
    wrappedReader.MoveToContent();
    wrappedReader.ReadEndElement();
    this.tokenStream = wrappedReader.XmlTokens;
    if (this.signature != null)
    {
        this.VerifySignature(this.signature, this.verificationKey);
    }
    this.BuildCryptoList();
}
```

`this.verificationKey` is used for signature verification. Let's see how it is resolved:

```
// System.IdentityModel.Tokens.SamlAssertion. Code ref: 14
protected void ReadSignature(XmlDictionaryReader reader, SecurityTokenSerializer
keyInfoSerializer, SecurityTokenResolver outOfBandTokenResolver, SamlSerializer samlSerializer)
{
    ...
    SignedXml signedXml = new SignedXml(new
StandardSignedInfo(samlSerializer.DictionaryManager), samlSerializer.DictionaryManager,
keyInfoSerializer);
    signedXml.TransformFactory = ExtendedTransformFactory.Instance;
    signedXml.ReadFrom(xmlDictionaryReader);
    SecurityKeyIdentifier keyIdentifier = signedXml.Signature.KeyIdentifier;
    this.verificationKey = SamlSerializer.ResolveSecurityKey(keyIdentifier,
outOfBandTokenResolver);
    if (this.verificationKey == null)
    {
        throw DiagnosticUtility.ExceptionUtility.ThrowHelperError(new
SecurityTokenException(SR.GetString("SAMLUnableToResolveSignatureKey", new object[]
        {
            this.issuer
        })));
    }
    this.signature = signedXml;
    this.signingToken = SamlSerializer.ResolveSecurityToken(keyIdentifier,
outOfBandTokenResolver);
    ...
}
```

As we mentioned earlier, different code is used to resolve the Security Key (for signature verification) and the Security Token (for authenticating the signing party). `ResolveSecurityKey()` and `ResolveSecurityToken()` use different approaches to process the items from the KeyInfo element: ResolveSecurityKey() uses, what we call, a depth-first approach, where for each key identifier clause, that is, for each <KeyInfo/> entry, it tries every available resolver:

```
// System.IdentityModel.Tokens.SamlSerializer. Code ref: 13
internal static SecurityKey ResolveSecurityKey(SecurityKeyIdentifier ski, SecurityTokenResolver
tokenResolver)
{
    if (ski == null)
    {
        throw DiagnosticUtility.ExceptionUtility.ThrowHelperArgumentNull("ski");
    }
    if (tokenResolver != null)
    {
        for (int i = 0; i < ski.Count; i++)
        {
            SecurityKey result = null;
            if (tokenResolver.TryResolveSecurityKey(ski[i], out result))
            {
                return result;
            }
        }
    }
    if (ski.CanCreateKey)
    {
            return ski.CreateKey();
    }
    return null;
}
```

On the other hand, `ResolveSecurityToken()` uses a breadth-first approach, where it tries each resolving method with all key identifier clauses before moving on to the next one. This makes our attack even easier:

```
// System.IdentityModel.Tokens.SamlSerializer. Code ref: 13
internal static SecurityToken ResolveSecurityToken(SecurityKeyIdentifier ski,
SecurityTokenResolver tokenResolver)
{
    SecurityToken securityToken = null;
    if (tokenResolver != null)
    {
        tokenResolver.TryResolveToken(ski, out securityToken);
    }
    RsaKeyIdentifierClause rsaKeyIdentifierClause;
    if (securityToken == null && ski.TryFind<RsaKeyIdentifierClause>(out
rsaKeyIdentifierClause))
    {
        securityToken = new RsaSecurityToken(rsaKeyIdentifierClause.Rsa);
    }
    X509RawDataKeyIdentifierClause x509RawDataKeyIdentifierClause;
    if (securityToken == null && ski.TryFind<X509RawDataKeyIdentifierClause>(out
x509RawDataKeyIdentifierClause))
    {
        securityToken = new X509SecurityToken(new
X509Certificate2(x509RawDataKeyIdentifierClause.GetX509RawData()));
    }
    return securityToken;
}
```

Also, we can see that the code in these two methods is very different, which enables attacks even in the case of a strict `tokenResolver`. For example, we notice that ***RsaKeyIdentifierClause*** and ***X509RawDataKeyIdentifierClause*** are resolved by both `ResolveSecurityToken()` and `ResolveSecurityKey()` methods, but `ResolveSecurityKey()` offers an additional method to resolve the key; using the `ski.CreateKey()` code. We can use a key identifier clause that is neither an RSA Key nor an x509 raw data one, but that supports creating the key from the data provided in the key identifier clause. For example, if we use a ***BinarySecretKeyIdentifierClause*** with a symmetric key as the first key identifier clause, `ResolveSecurityKey()` can resolve it (because of the `ski.CreateKey()` code) and use it to verify the signature, however `ResolveSecurityToken()` will not be able to resolve it, because it first tries to resolve all key identifier clauses as RSA keys, and then does the same attempt to resolve them as x509 raw data identifiers. Therefore, if we add a ***X509RawDataKeyIdentifierClause*** as the second clause, with the certificate of a valid expected authority, `ResolveSecurityToken()` will successfully resolve it and emit a Security Token for this certificate, which will pass the signing party validation as well.

`ResolveSecurityKey()` will loop through all <KeyInfo/> elements and for each one, it will try ALL resolvers until one is successful. In the other hand, `ResolveSecurityToken()` will do the opposite, it will loop all resolvers and for each one, it will try ALL <KeyInfo/> elements until one is successful. Therefore, for a successful attack, we need to provide two different keys that will make these methods return different keys. For example, sending a ***BinarySecretKeyIdentifierClause*** first followed by the expected certificate will meet this requirement.

Let’s see what token resolver is used by default, that is, what is the type of `tokenResolver` variable in the code above:

```
// System.ServiceModel.Security.ReceiveSecurityHeader. Code ref: 15
public void Process(TimeSpan timeout, ChannelBinding channelBinding, ExtendedProtectionPolicy
extendedProtectionPolicy)
{
    ...
    this.universalTokenResolver = new SecurityHeaderTokenResolver(this);
    this.primaryTokenResolver = new SecurityHeaderTokenResolver(this);
    ...
    if (this.outOfBandTokenResolver == null)
    {
        this.combinedUniversalTokenResolver = this.universalTokenResolver;
        this.combinedPrimaryTokenResolver = this.primaryTokenResolver;
    }
    else
    {
        this.combinedUniversalTokenResolver = new
AggregateSecurityHeaderTokenResolver(this.universalTokenResolver, this.outOfBandTokenResolver);
        this.combinedPrimaryTokenResolver = new
AggregateSecurityHeaderTokenResolver(this.primaryTokenResolver, this.outOfBandTokenResolver);
    }
    ...
    inferenceEngine.ExecuteProcessingPasses(this, xmlDictionaryReader);
    ...
```

Here we are interested in `combinedPrimaryTokenResolver`, which may be either a `SecurityHeaderTokenResolver` instance or an `AggregateSecurityHeaderTokenResolver` instance combining `SecurityHeaderTokenResolver` and `outOfBandTokenResolver`. Let’s analyze the most restrictive case, which is the `AggregateSecurityHeaderTokenResolver` since it will try more resolvers for our key identifiers.

For the cases we analyzed (several WCF services and Exchange Web Services), `outOfBandTokenResolver` is an instance of `System.IdentityModel.Selectors.SecurityTokenResolver.SimpleTokenResolver`

Let’s see how it would work if `AggregateSecurityHeaderTokenResolver` handles our specially crafted <KeyInfo/> element containing a ***BinarySecretKeyIdentifierClause*** and an ***X509RawDataKeyIdentifierClause***:

```
// System.ServiceModel.Security.AggregateSecurityHeaderTokenResolver. Code ref: 15
protected override bool TryResolveSecurityKeyCore(SecurityKeyIdentifierClause
keyIdentifierClause, out SecurityKey key)
{
    key = null;
    bool flag = this.tokenResolver.TryResolveSecurityKey(keyIdentifierClause, false, out key);
    if (!flag)
    {
        flag = base.TryResolveSecurityKeyCore(keyIdentifierClause, out key);
    }
    if (!flag)
    {
        flag = SecurityUtils.TryCreateKeyFromIntrinsicKeyClause(keyIdentifierClause, this, out
key);
    }
    return flag;
}
```

`this.tokenResolver` (`SecurityHeaderTokenResolve`) and `base.TryResolveSecurityKeyCore` (`SimpleTokenResolver`) is not able to resolve the ***BinarySecretKeyIdentifierClause*** or an

RSAKeyIdentifierClause, and therefore the `AggregateSecurityHeaderTokenResolver` ends up creating a security key from it.
However, when resolving the token, a breadth-first approach is used, and therefore each resolving method tries all key identifier clauses before moving to the next one. `SimpleTokenResolver` fails to resolve the ***BinarySecretKeyIdentifierClause*** or ***RSAKeyIdentifierClause,*** but if we send a second ***X509RawDataKeyIdentifierClause*** clause, `SimpleTokenResolver` is able to resolve it since it supports this type of clauses. `SecurityHeaderTokenResolver` has similar logic for the case of `AggregateSecurityHeaderTokenResolver`, so it is vulnerable as well.

There is no product-specific code in this example, so we can conclude that most of the products that use WS-Security with support for SAML token authentication are vulnerable to this attack. In fact, this is the case for any non-claim-based WCF service, because the claim-based ones use Windows Identity Foundation (WIF) resolvers (see WIF section below).

The <KeyInfo/> element of a SAML Assertion looks similar to:

```
<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
    <trust:BinarySecret xmlns:trust="http://docs.oasis-open.org/ws-sx/ws-
trust/200512">rV4k606Cp3UU0qlvUluOww==</trust:BinarySecret>
    <X509Data>
        <X509Certificate>MIIDYDCCAkigAwIB...029oTSEsV4ArsQ==</X509Certificate>
    </X509Data>
</KeyInfo>
```

Microsoft Exchange Server

To verify the findings from the previous section, we tested a custom WCF service, but also looked for a Microsoft flagship product that uses non-claim-based WCF services to prove the impact of the attack. We discovered that Microsoft Exchange Server does use this type of WCF service, but in general, we cannot access any WS-Security-protected web services directly without first authenticating against the Exchange front-end EWS (Exchange Web Service) proxy web app.

Because we were interested in a pre-auth attack rather than a privilege escalation one, we researched the exposed services and discovered that the service mapped to "***/ews/exchange.asmx/wssecurity***" is special. The front-end EWS proxy forwards unauthenticated requests to the back-end app and relies on the authentication to the Back-End EWS application, which in this case uses "standard" WS-Security libraries. As we mentioned earlier, we can pass both token signature verification and authentication of signing party for this endpoint and therefore are able to craft any arbitrary SAML Token that allows us to successfully authenticate as any user in EWS.

Windows Identity Foundation (WIF)

There is a good description of Windows Identity Foundation (WIF) in Microsoft web [18]:

"Windows Identity Foundation 4.5 is a set of .NET Framework classes for implementing claims-based identity in your applications. By using it, you'll more easily reap the benefits of claims-aware applications and services. WIF 4.5 can be used in any Web application or Web service that uses the .NET Framework version 4.5 or later. WIF is just one part of Microsoft's Federated Identity software family that implements the shared industry vision based on open standards. Federated Identity comprises three components: Active Directory® Federation Services (AD

FS) 2.0, [Windows Azure Access Control Services](#) (ACS), and WIF. Together, these three components form the core of Microsoft's new claims-based cloud identity and access platform."

One way of exchanging WIF claims is to use SAML tokens where SAML assertion attributes are mapped with WIF claims. In this section, we review a default WIF implementation and point out where customized ones can differ.

First, SAML tokens are read in `System.IdentityModel.Tokens.SamlSecurityTokenHandler:`

```
// System.IdentityModel.Tokens.SamlSecurityTokenHandler. Code ref: 16
public override SecurityToken ReadToken(XmlReader reader)
{
    ...
    SamlAssertion samlAssertion = this.ReadAssertion(reader);
    SecurityToken signingToken;
    this.TryResolveIssuerToken(samlAssertion, base.Configuration.IssuerTokenResolver, out
signingToken);
    samlAssertion.SigningToken = signingToken;
    return new SamlSecurityToken(samlAssertion);
}
```

Here we are interested in two calls: `ReadAssertion(reader)` and `TryResolveIssuerToken()`.

```
// System.IdentityModel.Tokens.SamlSecurityTokenHandler. Code ref: 16
protected virtual SamlAssertion ReadAssertion(XmlReader reader)
{
    ...
    SamlAssertion samlAssertion = new SamlAssertion();
    EnvelopedSignatureReader envelopedSignatureReader = new EnvelopedSignatureReader(reader,
new SamlSecurityTokenHandler.WrappedSerializer(this, samlAssertion),
base.Configuration.IssuerTokenResolver, false, true, false);
    ...
    envelopedSignatureReader.ReadEndElement();
    samlAssertion.SigningCredentials = envelopedSignatureReader.SigningCredentials;
    samlAssertion.CaptureSourceData(envelopedSignatureReader);
    return samlAssertion;
}
```

As we can see, it uses `EnvelopedSignatureReader` with `resolveIntrinsicSigningKeys=false` for signature verification.

Let's take a look at how it resolves security keys for this verification:

```
// System.IdentityModel.EnvelopedSignatureReader. Code ref: 17
private void ResolveSigningCredentials()
{
    ...
    SecurityKey signingKey = null;
    if
(!this._signingTokenResolver.TryResolveSecurityKey(this._signedXml.Signature.KeyIdentifier[0],
out signingKey))
    ...
    WifSignedInfo wifSignedInfo = this._signedXml.Signature.SignedInfo as WifSignedInfo;
    this._signingCredentials = new SigningCredentials(signingKey,
this._signedXml.Signature.SignedInfo.SignatureMethod, wifSignedInfo[0].DigestMethod,
this._signedXml.Signature.KeyIdentifier);
}
```

There is something here that is critical for our attack: it **only tries to resolve the first key identifier clause** using `signingTokenResolver.TryResolveSecurityKey`(), and since `resolveIntrinsicSigningKeys=false`, it will not try to create a key.

At the end of this method, it saves `SigningCredentials` for later use, but it still contains all ***key identifiers*** (last parameter), not just the first one that was used for resolving the key:

```
// System.IdentityModel.Tokens.SigningCredentials. Code ref: 18
public SigningCredentials(SecurityKey signingKey, string signatureAlgorithm, string
digestAlgorithm, SecurityKeyIdentifier signingKeyIdentifier)
{
    ...
    this.signingKey = signingKey;
    this.signatureAlgorithm = signatureAlgorithm;
    this.digestAlgorithm = digestAlgorithm;
    this.signingKeyIdentifier = signingKeyIdentifier;
}
```

Now let's go back to `System.IdentityModel.Tokens.SamlSecurityTokenHandler.ReadToken()` and review the `TryResolveIssuerToken()` call:

```
// System.IdentityModel.Tokens.SamlSecurityTokenHandler. Code ref: 16
protected virtual bool TryResolveIssuerToken(SamlAssertion assertion, SecurityTokenResolver
issuerResolver, out SecurityToken token)
{
    ...
    if (assertion.SigningCredentials != null &&
assertion.SigningCredentials.SigningKeyIdentifier != null && issuerResolver != null)
    {
        SecurityKeyIdentifier signingKeyIdentifier =
assertion.SigningCredentials.SigningKeyIdentifier;
        return issuerResolver.TryResolveToken(signingKeyIdentifier, out token);
    }
    ...
}
```

As we can see, it tries to resolve the security token using `issuerResolver.TryResolveToken()` for **all *key identifiers***.

Now that we know this critical difference, let's review the resolver that is used in both cases: `System.IdentityModel.Tokens.IssuerTokenResolver`.

```
// System.IdentityModel.Tokens.IssuerTokenResolver. Code ref: 19
protected override bool TryResolveSecurityKeyCore(SecurityKeyIdentifierClause
keyIdentifierClause, out SecurityKey key)
{
    if (keyIdentifierClause == null)
    {
        throw
DiagnosticUtility.ExceptionUtility.ThrowHelperArgumentNull("keyIdentifierClause");
    }
    key = null;
    X509RawDataKeyIdentifierClause x509RawDataKeyIdentifierClause = keyIdentifierClause as
X509RawDataKeyIdentifierClause;
    if (x509RawDataKeyIdentifierClause != null)
    {
        key = x509RawDataKeyIdentifierClause.CreateKey();
        return true;
    }
    RsaKeyIdentifierClause rsaKeyIdentifierClause = keyIdentifierClause as
RsaKeyIdentifierClause;
    if (rsaKeyIdentifierClause != null)
    {
        key = rsaKeyIdentifierClause.CreateKey();
        return true;
    }
    return this._wrappedTokenResolver.TryResolveSecurityKey(keyIdentifierClause, out key);
}
```

```
// System.IdentityModel.Tokens.IssuerTokenResolver. Code ref: 19
protected override bool TryResolveTokenCore(SecurityKeyIdentifierClause keyIdentifierClause,
out SecurityToken token)
{
    if (keyIdentifierClause == null)
    {
        throw
DiagnosticUtility.ExceptionUtility.ThrowHelperArgumentNull("keyIdentifierClause");
    }
    token = null;
    X509RawDataKeyIdentifierClause x509RawDataKeyIdentifierClause = keyIdentifierClause as
X509RawDataKeyIdentifierClause;
    if (x509RawDataKeyIdentifierClause != null)
    {
        token = new X509SecurityToken(new
X509Certificate2(x509RawDataKeyIdentifierClause.GetX509RawData()));
        return true;
    }
    RsaKeyIdentifierClause rsaKeyIdentifierClause = keyIdentifierClause as
RsaKeyIdentifierClause;
    if (rsaKeyIdentifierClause != null)
    {
        token = new RsaSecurityToken(rsaKeyIdentifierClause.Rsa);
        return true;
    }
    return this._wrappedTokenResolver.TryResolveToken(keyIdentifierClause, out token);
}
```

As we can see, both work in a similar way for the two main cases: ***x509RawDataKeyIdentifierClause*** and ***RsaKeyIdentifierClause,*** and they return similar results. However, any other key identifier clause type is handled by `_wrappedTokenResolver`. Let's take a closer look:

```
// System.IdentityModel.Tokens.IssuerTokenResolver. Code ref: 19
public class IssuerTokenResolver : SecurityTokenResolver
{
    public static readonly StoreName DefaultStoreName = StoreName.TrustedPeople;
    public static readonly StoreLocation DefaultStoreLocation = StoreLocation.LocalMachine;
    private SecurityTokenResolver _wrappedTokenResolver;

    public IssuerTokenResolver() : this(new
X509CertificateStoreTokenResolver(IssuerTokenResolver.DefaultStoreName,
IssuerTokenResolver.DefaultStoreLocation))
    {
    }

    public IssuerTokenResolver(SecurityTokenResolver wrappedTokenResolver)
    {
    ...
    this._wrappedTokenResolver = wrappedTokenResolver;
    }
```

In default configuration wrapped token resolver is a `System.IdentityModel.Tokens.X509CertificateStoreTokenResolver` instance with its certificate storage set to `StoreName.TrustedPeople` on `StoreLocation.LocalMachine`. It is used for resolving x509 certificates from that specific storage (in our case, Trusted People on Local Machine):

```
// System.IdentityModel.Tokens.X509CertificateStoreTokenResolver. Code ref: 20
protected override bool TryResolveTokenCore(SecurityKeyIdentifierClause keyIdentifierClause,
out SecurityToken token)
{
    if (keyIdentifierClause == null)
    {
        throw
DiagnosticUtility.ExceptionUtility.ThrowHelperArgumentNull("keyIdentifierClause");
    }
    token = null;
    X509Store x509Store = null;
    X509Certificate2Collection x509Certificate2Collection = null;
    try
    {
        x509Store = new X509Store(this.storeName, this.storeLocation);
        x509Store.Open(OpenFlags.ReadOnly);
        x509Certificate2Collection = x509Store.Certificates;
        X509Certificate2Enumerator enumerator = x509Certificate2Collection.GetEnumerator();
        while (enumerator.MoveNext())
        {
            X509Certificate2 current = enumerator.Current;
           X509ThumbprintKeyIdentifierClause x509ThumbprintKeyIdentifierClause =
keyIdentifierClause as X509ThumbprintKeyIdentifierClause;
            if (x509ThumbprintKeyIdentifierClause != null &&
x509ThumbprintKeyIdentifierClause.Matches(current))
            {
                token = new X509SecurityToken(current);
                bool result = true;
                return result;
            }
            X509IssuerSerialKeyIdentifierClause x509IssuerSerialKeyIdentifierClause =
keyIdentifierClause as X509IssuerSerialKeyIdentifierClause;
            if (x509IssuerSerialKeyIdentifierClause != null &&
x509IssuerSerialKeyIdentifierClause.Matches(current))
            {
                token = new X509SecurityToken(current);
                bool result = true;
                 return result;
            }
            X509SubjectKeyIdentifierClause x509SubjectKeyIdentifierClause = keyIdentifierClause
as X509SubjectKeyIdentifierClause;
            if (x509SubjectKeyIdentifierClause != null &&
x509SubjectKeyIdentifierClause.Matches(current))
            {
                token = new X509SecurityToken(current);
                bool result = true;
                return result;
            }
            X509RawDataKeyIdentifierClause x509RawDataKeyIdentifierClause = keyIdentifierClause
as X509RawDataKeyIdentifierClause;
            if (x509RawDataKeyIdentifierClause != null &&
x509RawDataKeyIdentifierClause.Matches(current))
            {
                token = new X509SecurityToken(current);
                bool result = true;
                return result;
            }
        }
    }
    finally
    {
        if (x509Certificate2Collection != null)
        {
            for (int i = 0; i < x509Certificate2Collection.Count; i++)
            {
                x509Certificate2Collection[i].Reset();
            }
        }
        if (x509Store != null)
        {
            x509Store.Close();
        }
    }
    return false;
}
```

Let’s have a look at how it resolves security keys:

```
// System.IdentityModel.Tokens.X509CertificateStoreTokenResolver. Code ref: 20
protected override bool TryResolveSecurityKeyCore(SecurityKeyIdentifierClause
keyIdentifierClause, out SecurityKey key)
{
    if (keyIdentifierClause == null)
    {
        throw
DiagnosticUtility.ExceptionUtility.ThrowHelperArgumentNull("keyIdentifierClause");
    }
    key = null;
    EncryptedKeyIdentifierClause encryptedKeyIdentifierClause = keyIdentifierClause as
EncryptedKeyIdentifierClause;
    if (encryptedKeyIdentifierClause != null)
    {
        SecurityKeyIdentifier encryptingKeyIdentifier =
encryptedKeyIdentifierClause.EncryptingKeyIdentifier;
        if (encryptingKeyIdentifier != null && encryptingKeyIdentifier.Count > 0)
        {
            for (int i = 0; i < encryptingKeyIdentifier.Count; i++)
            {
                SecurityKey securityKey = null;
                if (base.TryResolveSecurityKey(encryptingKeyIdentifier[i], out securityKey))
                {
                    byte[] encryptedKey = encryptedKeyIdentifierClause.GetEncryptedKey();
                    string encryptionMethod = encryptedKeyIdentifierClause.EncryptionMethod;
                    byte[] symmetricKey = securityKey.DecryptKey(encryptionMethod,
encryptedKey);
                    key = new InMemorySymmetricSecurityKey(symmetricKey, false);
                    return true;
                }
            }
        }
    }
    else
    {
        SecurityToken securityToken = null;
        if (base.TryResolveToken(keyIdentifierClause, out securityToken) &&
securityToken.SecurityKeys.Count > 0)
       {
            key = securityToken.SecurityKeys[0];
            return true;
        }
    }
    return false;
}
```

The second part is used to resolve x509 certs from specific storage and return the security keys. However, the first part (highlighted in yellow) is used to handle ***encryptedKeyIdentifierClause***, which as we saw above, is not supported by `TryResolveTokenCore()` and therefore it will not be able to resolve them. This is what we need for a successful attack, but there is one requirement: it uses the Private Key of the x509 certificate for decryption of the ***encryptedKeyIdentifierClause.***

Let’s imagine the next vulnerable environment:

- Target server has an x509 certificate with Private Key in ***Trusted People*** on ***Local Machine*** storage

- An attacker has access to the Public Key of such certificate and uses it for encrypting a Symmetric Key used for signature calculation.
- `IssuerTokenResolver` does not support **encryptedKeyIdentifierClause,** so it is handled by the wrapped resolver **X509CertificateStoreTokenResolver**. Symmetric Key is decrypted and signature verification is passed.
- But **encryptedKeyIdentifierClause** cannot be resolved for the security token since neither `IssuerTokenResolver` nor `X509CertificateStoreTokenResolver` supports them, so when the server tries to resolve token for signing party verification, the **encryptedKeyIdentifierClause** is skipped and the next key identifier clause in the KeyInfo section is taken. Attacker can send the expected key for successful verification.

Despite the fact that the requirement of having access to the public key of an X509 certificate with Private Key stored in the ***Local Machine/Trusted People*** storage significantly decreases the chances of carrying out this attack, we need to take into account the fact that the number of products and servers using Windows Identity Foundation (WIF) is enormous, and thus potentially there are servers that can be attacked this way.

Note that this vector is valid for any application using Windows Identity Foundation (WIF) including claim-based WCF services. It does not matter where the token is generated (for example, Azure Active Directory, ADFS, or other).

The <KeyInfo/> element of SAML Assertion looks similar to:

```
<ds:KeyInfo>
    <xenc:EncryptedKey xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
           <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
                  <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
                     <ds:X509Data>
                            <ds:X509Certificate>….</ds:X509Certificate>
                     </ds:X509Data>
                  </ds:KeyInfo>
                  <xenc:CipherData>
                         <xenc:CipherValue>e++….</xenc:CipherValue>
                  </xenc:CipherData>
           </xenc:EncryptedKey>
           <ds:X509Data>
                  <ds:X509Certificate>MIIDBTCCAeS1IbC...Cf6zzzWh</ds:X509Certificate>
              </ds:X509Data>
</ds:KeyInfo>
```

Microsoft SharePoint

In the previous section, we reviewed the case of the default WIF configuration, that is, using the default security token resolvers and their default configuration. However, applications can use a different configuration (for example, different certificate storage) or even use a custom security token resolver. A good example of such a case is SharePoint server, which uses its own resolver:
`Microsoft.SharePoint.IdentityModel.SPIssuerTokenResolver.`

The process of SAML parsing and validation is similar to the previous case with WIF. The only difference is in the security token resolver:

```
// Microsoft.SharePoint.IdentityModel.SPIssuerTokenResolver
protected override bool TryResolveSecurityKeyCore(SecurityKeyIdentifierClause
keyIdentifierClause, out SecurityKey key)
{
    key = null;
    EncryptedKeyIdentifierClause encryptedKeyIdentifierClause = keyIdentifierClause as
EncryptedKeyIdentifierClause;
    if (encryptedKeyIdentifierClause != null)
    {
            ...
    }
    else
    {
             SecurityToken securityToken = null;
             if (base.TryResolveToken(keyIdentifierClause, out securityToken))
             {
                 ...
             }
             else if (keyIdentifierClause.CanCreateKey)
             {
            key = keyIdentifierClause.CreateKey();
                ...
            return true;
             }
    }
    ...
    return false;
}
```

For a successful attack, we must reach the green code above. Therefore, the first key clause should not be resolved by `TryResolveToken()` and should be able to pass the `CreateKey` check. An RSA public key, for example, satisfies both requirements, and we can use it for signature calculation.

Now let’s see how the security token is resolved:

```
// Microsoft.SharePoint.IdentityModel.SPIssuerTokenResolver
protected override bool TryResolveTokenCore(SecurityKeyIdentifier keyIdentifier, out
SecurityToken token)
{
    …
    token = null;
    foreach (SecurityKeyIdentifierClause current in keyIdentifier)
    {
        if (base.TryResolveToken(current, out token)) {
            ...
            return true;
        }
    }
    ...
    return false;
}
```

The code tries to resolve all key clauses one by one and, as we already mentioned, an RSA public key cannot be resolved here, so it tries to resolve the security token using the remaining key clauses:

```
// Microsoft.SharePoint.IdentityModel.SPIssuerTokenResolver
protected override bool TryResolveTokenCore(SecurityKeyIdentifierClause keyIdentifierClause,
out SecurityToken token)
```

```
{
    ...
    token = null;
    SPSecurityTokenServiceManager local = SPSecurityTokenServiceManager.Local;
    foreach (SPTrustedLoginProvider current in local.TrustedLoginProviders)
    {
        if (this.TryResolveTokenCoreWithAccessProvider(current, keyIdentifierClause, out
token))
        {
            ...
            bool result = true;
            return result;
        }
    }
    foreach (SPLoginProviderBase current2 in local.TrustedSecurityTokenServices)
    {
        ...
        else if (this.TryResolveTokenCoreWithAccessProvider(current2, keyIdentifierClause, out
token))
        {
            ...
            bool result = true;
            return result;
        }
    }
    if (this.TryResolveTokenCoreWithAccessProvider(local.LocalLoginProvider,
keyIdentifierClause, out token))
    {
        ...
        return true;
    }
    ...
    return false;
}
```

Similarly, to the Exchange server and WCF cases, an attacker can use its own RSA private key to sign the tampered SAML Assertion and include the RSA public key as the first key identifier clause, so this assertion passes signature verification. On the other hand, an attacker can add an additional key identifier clause, for example an x509 Thumbprint of expected signature authority, and this one will be resolved as a security token, so the SAML token passes signing party verification as well. The <KeyInfo/> element of SAML Assertion looks similar to:

```
<KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
    <KeyValue>
            <RSAKeyValue
                  <Modulus>rNZGlF…jQbiKgUaK3Ir80VU=</Modulus>
                  <Exponent>AQAB</Exponent>
            </RSAKeyValue>
    </KeyValue>
    <o:SecurityTokenReference xmlns:o="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd">
            <o:KeyIdentifier ValueType="http://docs.oasis-open.org/wss/oasis-wss-soap-message-
security-1.1#ThumbprintSHA1">w6PEGcLSM0WP0fStydLujiYGZZA=</o:KeyIdentifier>
    </o:SecurityTokenReference>
</KeyInfo>
```

As a result, the attacker can arbitrarily generate or tamper with any SAML assertions to authenticate himself as any desired user.

Conclusion

In this paper, we described not only the details of the vulnerabilities we found, but also the mental process we used to find them. Our intent is not only to alert the audience of the impact of these vulnerabilities and urge operations teams to patch their systems quickly, but also to show developers how critical parts of popular vendors' code can become vulnerable because of inconsistent ways of parsing the same data. Similar inconsistencies can occur in other places of your codebase and we hope we are able to show you how to look for them.

Our findings presented in the Arbitrary Constructor Invocation section show that even instantiation of arbitrary Types can be very dangerous and depend on "gadgets" available on the system. To avoid this type of problem, developers must limit the amount of Types that can processed as much as possible. They should also be very careful when implementing methods that can be used as gadget-like constructors and custom assembly resolvers.

The root problem for Dupe Key Confusion Attacks is not in SAML tokens, but in how XML signatures [19] are verified, which means that any other place where an XML signature is used might be affected. For example, attackers can tamper with a federation metadata document, aka FederatedMetadata.xml. Also, note that WS-Security does not only use XML Signatures for SAML – SOAP signatures and potentially encryption may be target of this attack as well.

Even though these vulnerabilities were found in libraries related to SAML or JWT, we consider these protocols to be secure and both of them can be used for delegated authentication if they are properly implemented and configured. However, the devil is in the details, and in this case, in the implementation. SAML is a complex protocol, and its secure implementation is definitely not easy task. Although it has been in existence for many years, security researchers continue to find flaws in its implementation.

Despite the fact that the presented bugs were found in .NET frameworks, we do not consider these attacks to be .Net specific problems. The XML signature standard [19] allows many items in the <KeyInfo /> element and any inconstancy in how they are used for signature validation and authentication of signing party may allow the described attack regardless of the frameworks or languages of implementation.

References

1. **Federated Identities: A Developer's Primer** [https://hackernoon.com/federated-identities-a-developers-primer-655a160d66cb]
2. **CVE-2019-1083** [https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/CVE-2019-1083]
3. **JSON Web Tokens** [https://en.wikipedia.org/wiki/JSON_Web_Token]
4. **SAML Specification** [http://saml.xml.org/saml-specifications]
5. **JSON Web Signature (JWS)** [https://tools.ietf.org/html/rfc7515]
6. **JSON Web Encryption (JWE)** [https://tools.ietf.org/html/rfc7516]
7. **System.IdentityModel.Tokens.Jwt** [https://www.nuget.org/packages/System.IdentityModel.Tokens.Jwt]
8. **Resolving Assembly Loads** [https://docs.microsoft.com/en-us/dotnet/framework/app-domains/resolve-assembly-loads]
9. **CVE-2019-1006** [https://portal.msrc.microsoft.com/en-us/security-guidance/advisory/CVE-2019-1006]
10. **WS-Federation Protocol** [http://docs.oasis-open.org/wsfed/federation/v1.2/os/ws-federation-1.2-spec-os.html]
11. **SAML Protocol** [http://saml.xml.org/protocols]
12. **Federated Identity pattern** [https://docs.microsoft.com/en-us/azure/architecture/patterns/federated-identity]
13. **An Introduction to XML Digital Signatures** [https://www.xml.com/pub/a/2001/08/08/xmldsig.html]
14. **On Breaking SAML: Be Whoever You Want to Be** [https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final91-8-23-12.pdf]
15. **Duo Finds SAML Vulnerabilities Affecting Multiple Implementations** [https://duo.com/blog/duo-finds-saml-vulnerabilities-affecting-multiple-implementations]
16. **XML Signature Wrapping** [https://www.ws-attacks.org/XML_Signature_Wrapping]
17. **What Is Windows Communication Foundation** [https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf]
18. **Windows Identity Foundation 4.5 Overview** [https://docs.microsoft.com/es-es/dotnet/framework/security/wif-overview]
19. **XML Signature Syntax and Processing** [https://www.w3.org/TR/xmldsig-core/]

Code References

1. https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/blob/master/src/System.IdentityModel.Tokens.Jwt/JwtSecurityTokenHandler.cs
2. https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/blob/master/src/Microsoft.IdentityModel.Tokens/SignatureProvider.cs
3. https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/blob/master/src/Microsoft.IdentityModel.Tokens/AsymmetricSignatureProvider.cs
4. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/Tokens/X509AsymmetricSecurityKey.cs
5. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/CryptoHelper.cs
6. https://github.com/microsoft/referencesource/blob/master/mscorlib/system/security/cryptography/cryptoconfig.cs
7. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/Tokens/SamlAssertion.cs
8. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/SignedXml.cs
9. https://github.com/microsoft/referencesource/blob/master/System.Web.Mobile/Mobile/CookielessData.cs
10. https://github.com/microsoft/referencesource/blob/master/System.Web/Security/PassportIdentity.cs
11. https://github.com/PowerShell/PowerShell/blob/master/src/System.Management.Automation/engine/remoting/fanin/WSManPluginFacade.cs
12. https://github.com/microsoft/referencesource/blob/master/mscorlib/system/runtime/interopservices/gchandle.cs
13. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/Tokens/SamlSerializer.cs
14. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/Tokens/SamlAssertion.cs
15. https://github.com/microsoft/referencesource/blob/master/System.ServiceModel/System/ServiceModel/Security/ReceiveSecurityHeader.cs
16. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/Tokens/SamlSecurityTokenHandler.cs
17. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/EnvelopedSignatureReader.cs
18. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/Tokens/SigningCredentials.cs
19. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/Tokens/IssuerTokenResolver.cs
20. https://github.com/microsoft/referencesource/blob/master/System.IdentityModel/System/IdentityModel/Tokens/X509CertificateStoreTokenResolver.cs