SSO Wars

The Token Menace

> whoarewe

- **Alvaro Muñoz**
  Security Researcher with Micro Focus Fortify team
  @Pwntester

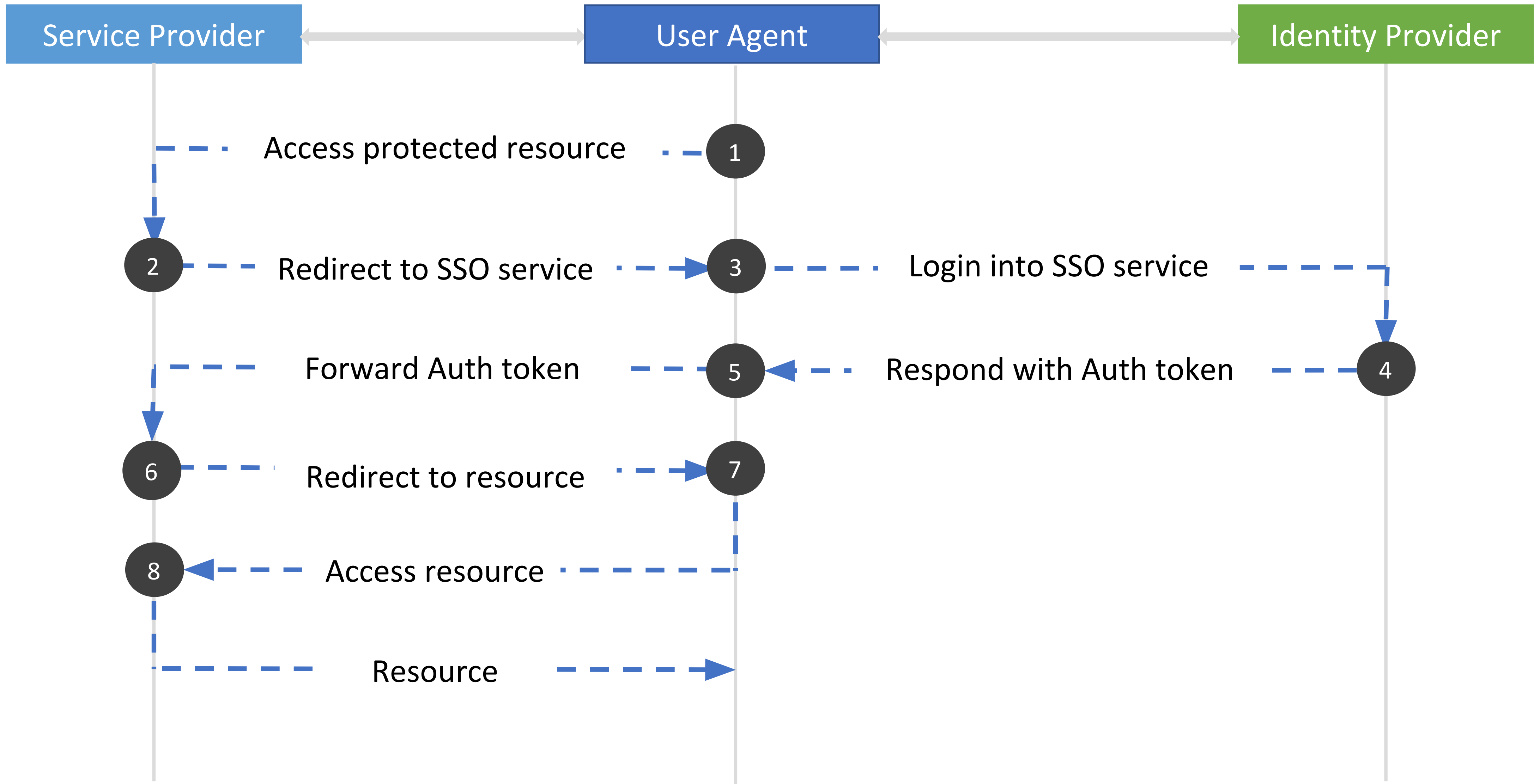
- **Oleksandr Mirosh**
  Security Researcher with Micro Focus Fortify team
  @OlekMirosh

Agenda

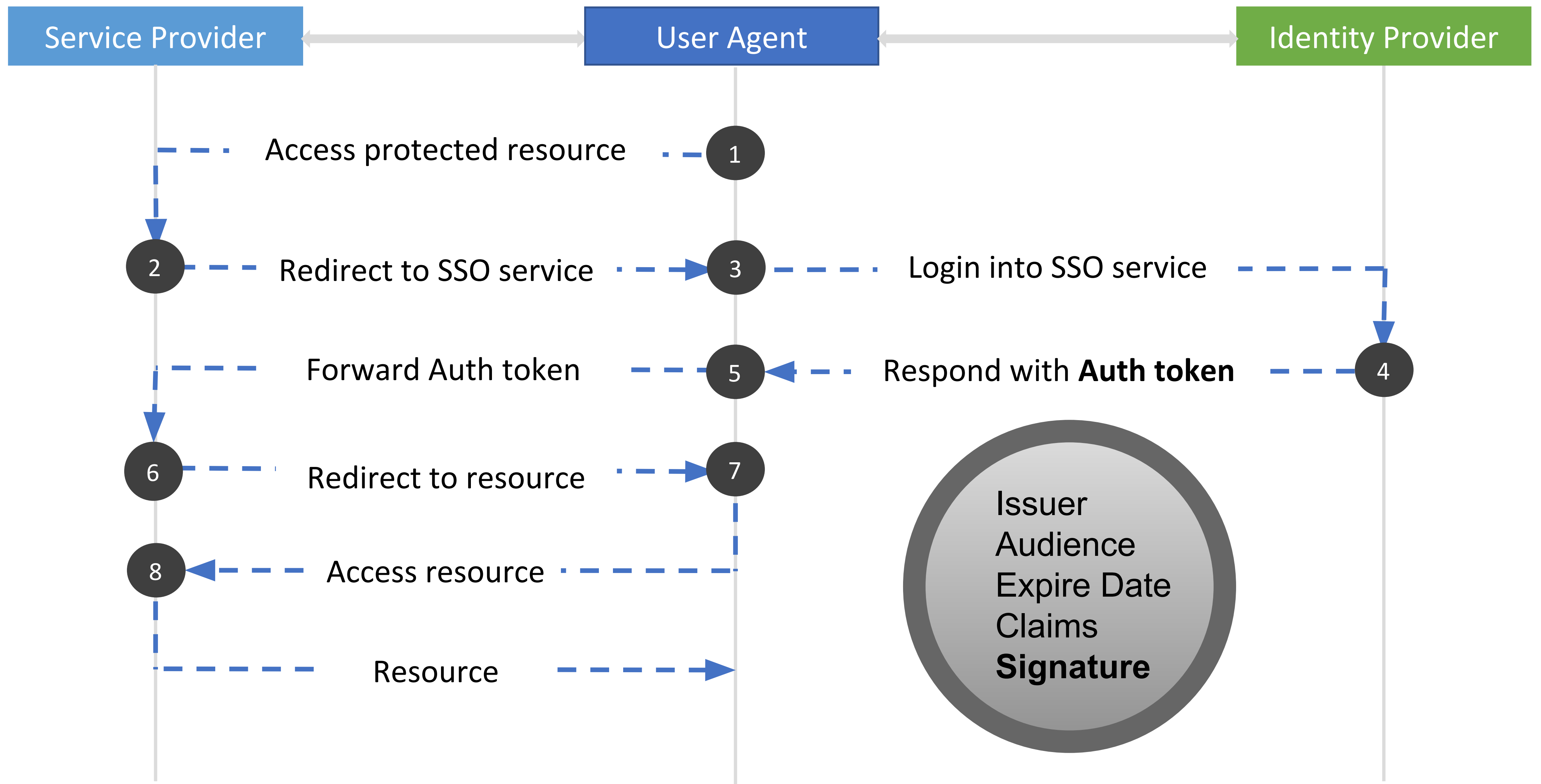
- Introduction
  - Authentication Tokens
  - Delegated Authentication
- Arbitrary Constructor Invocation
  - Potential attack vectors
- Dupe Key Confusion
  - Windows Communication Foundation (WCF)
  - Windows Identity Foundation (WIF)
- Conclusions

Introduction

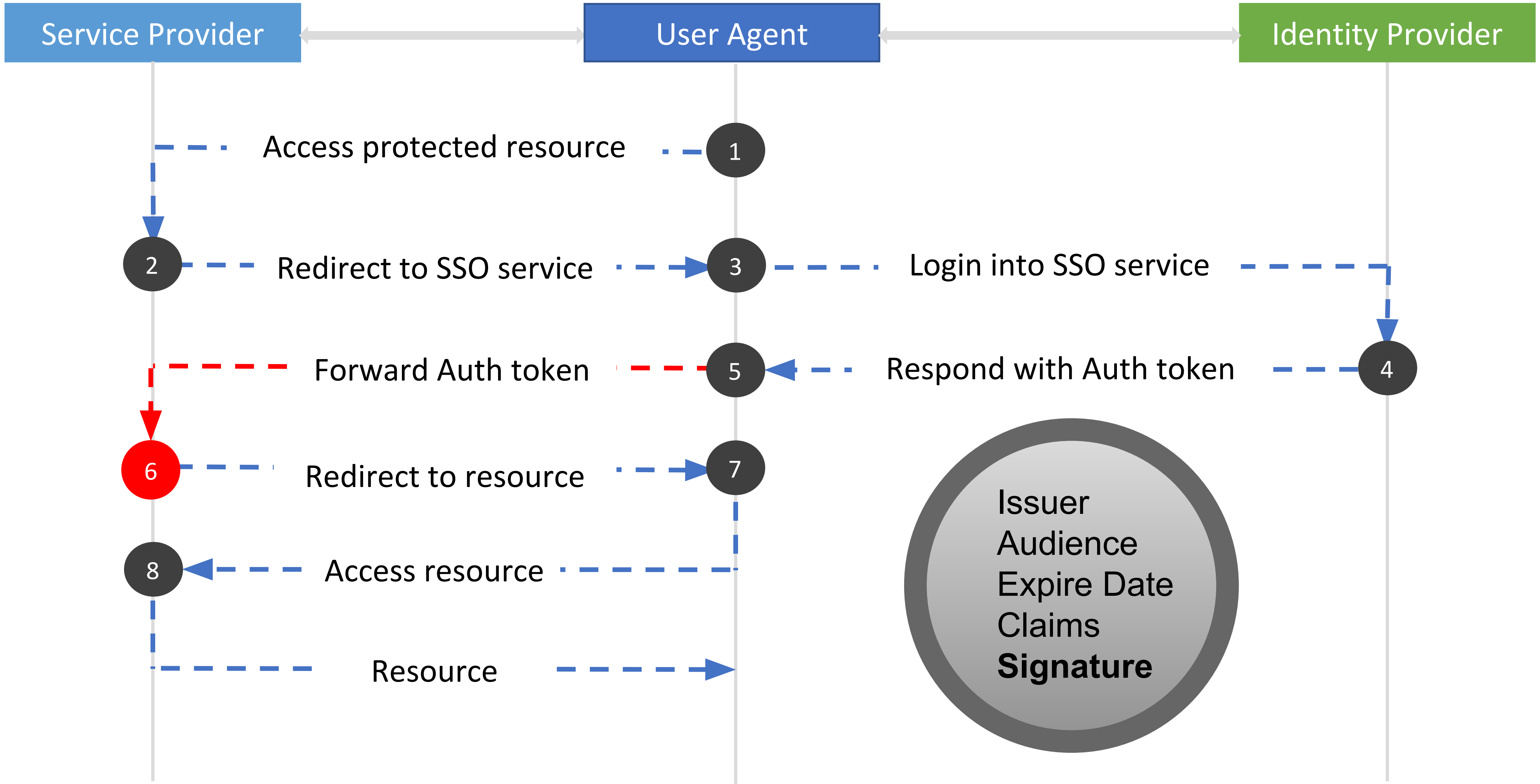
Delegated Authentication



Delegated Authentication



Delegated Authentication



Potential attack vectors

Token parsing vulnerabilities

Normally **before** signature verification

Attack Token parsing process

Eg: CVE-2019-1083

Signature verification bypasses

The holy grail

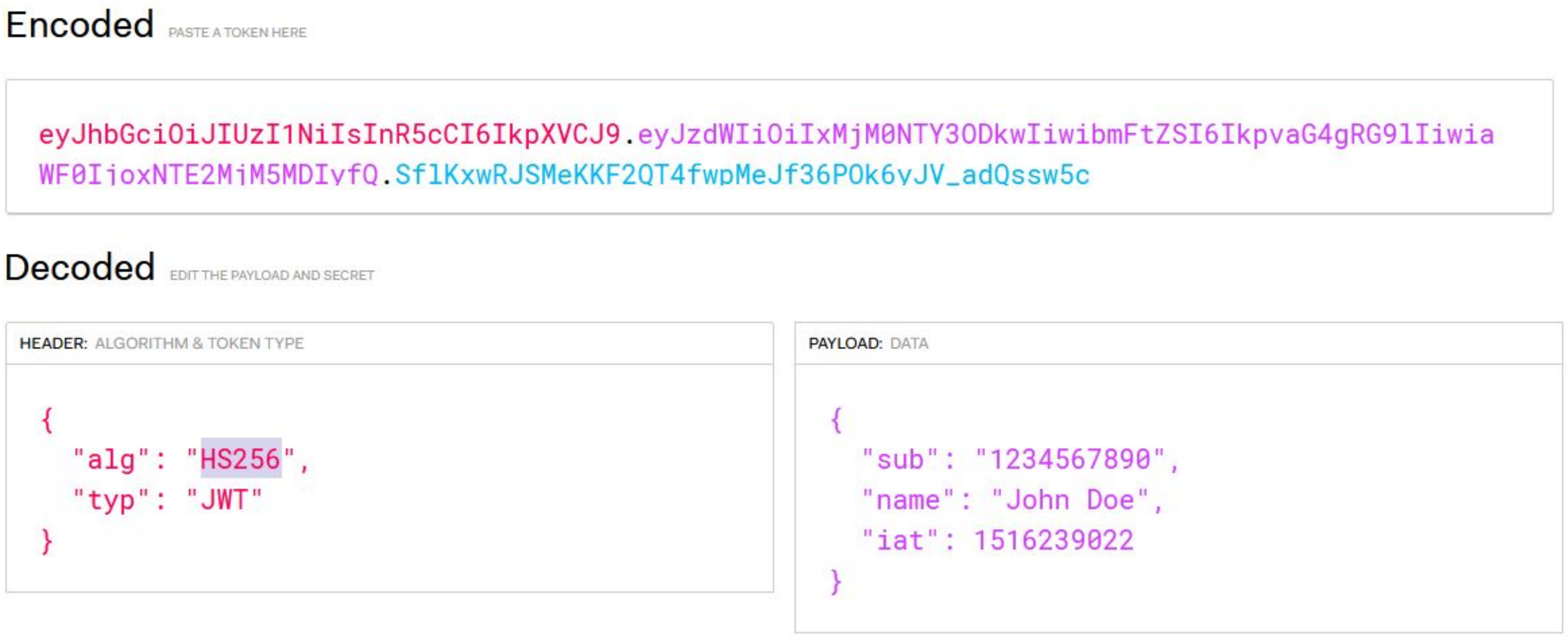
Enable us to tamper claims in the token

Eg: CVE-2019-1006

Arbitrary Constructor Invocation

CVE-2019-1083

JWT token



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwia
WF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Source: http://jwt.io

System.IdentityModel.Tokens.Jwt library

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

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

```
// System.Security.Cryptography.CryptoConfig
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;
    ...
    MethodBase[] array = runtimeType.GetConstructors(BindingFlags.Instance | BindingFlags.Public |
BindingFlags.CreateInstance);
    ...
    object obj;
    RuntimeConstructorInfo runtimeConstructorInfo = Type.DefaultBinder.BindToMethod(BindingFlags.Instance |
BindingFlags.Public | BindingFlags.CreateInstance, array, ref args, null, null, null, out obj)
     ...
    object result = runtimeConstructorInfo.Invoke(BindingFlags.Instance | BindingFlags.Public |
BindingFlags.CreateInstance, Type.DefaultBinder, args, null);
```

Similar code for SAML

```
<saml:Assertion ...>
        ...
        <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
        <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
             <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
              ...
        </ds:SignedInfo>
        <ds:SignatureValue>WNKeaE3R....SLMRLfIN/zI=</ds:SignatureValue>
        ...
    </ds:Signature>
</saml:Assertion>
```

```
// System.IdentityModel.SignedXml
public void StartSignatureVerification(SecurityKey verificationKey) {
        string signatureMethod = this.Signature.SignedInfo.SignatureMethod;
        ...
        using (HashAlgorithm hash = asymmetricKey.GetHashAlgorithmForSignature(signatureMethod))
        ...
```

- YAY! We can call public parameterless constructor
  - Doesn’t sound too exciting or does it?
- We actually control some data:
  - The name of the type to be resolved
  - Request’s parameters, cookies, headers, etc.
    - In .NET the request is accessed through a static property. E.g.:

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

Potential Attack Vectors (1/2)

- Information Leakage
- *For example:* SharePoint server returns different results when Type resolution and instantiation was successful or not. These results may enable an attacker to collect information about available libraries and products on the target server.
- Denial of Service
    - We found gadgets that trigger an Unhandled Exception. They enable an attacker to leave SharePoint server unresponsive for a period of time.

Potential Attack Vectors (2/2)

- Arbitrary Code Execution
  - We can search for a gadget that installs an insecure assembly resolver on its static constructor
  - We can then send full-qualified type name (including assembly name) which:
    - Not available in the GAC, the system will fall back to resolving it using insecure assembly resolver
    - Insecure assembly resolver will load the assembly and then instantiate the type
  - Downside:
    - May depend on server configurations, e.g. already enabled *AssemblyResolvers*
    - May require ability to upload malicious *dll* to the server ☹️

First payload: Microsoft.Exchange.Search.Fast.FastManagementClient

```
// Microsoft.Exchange.Search.Fast.FastManagementClient
static FastManagementClient() {
    ...
    AppDomain.CurrentDomain.AssemblyResolve += new ResolveEventHandler(OnAssemblyResolveEvent);
}
```

```
// Microsoft.Exchange.Search.Fast.FastManagementClient
private static Assembly OnAssemblyResolveEvent(object sender, ResolveEventArgs args) {
    string name = args.Name.Split(new char[]{','})[0];
    string path1 = Path.Combine(FastManagementClient.fsisInstallPath, "Installer\\Bin");
    string path2 = Path.Combine(FastManagementClient.fsisInstallPath, "HostController");
    string[] paths = new string[]   {path1,path2};
    for (int i = 0; i < paths.Length; i++) {
        string full_path = paths[i] + Path.DirectorySeparatorChar.ToString() + name + ".dll";
        if (File.Exists(full_path)) return Assembly.LoadFrom(full_path);
      ...
```

Second payload: ..\..\..\..\..\..\..\tmp\malicious

Demo

Exchange RCE

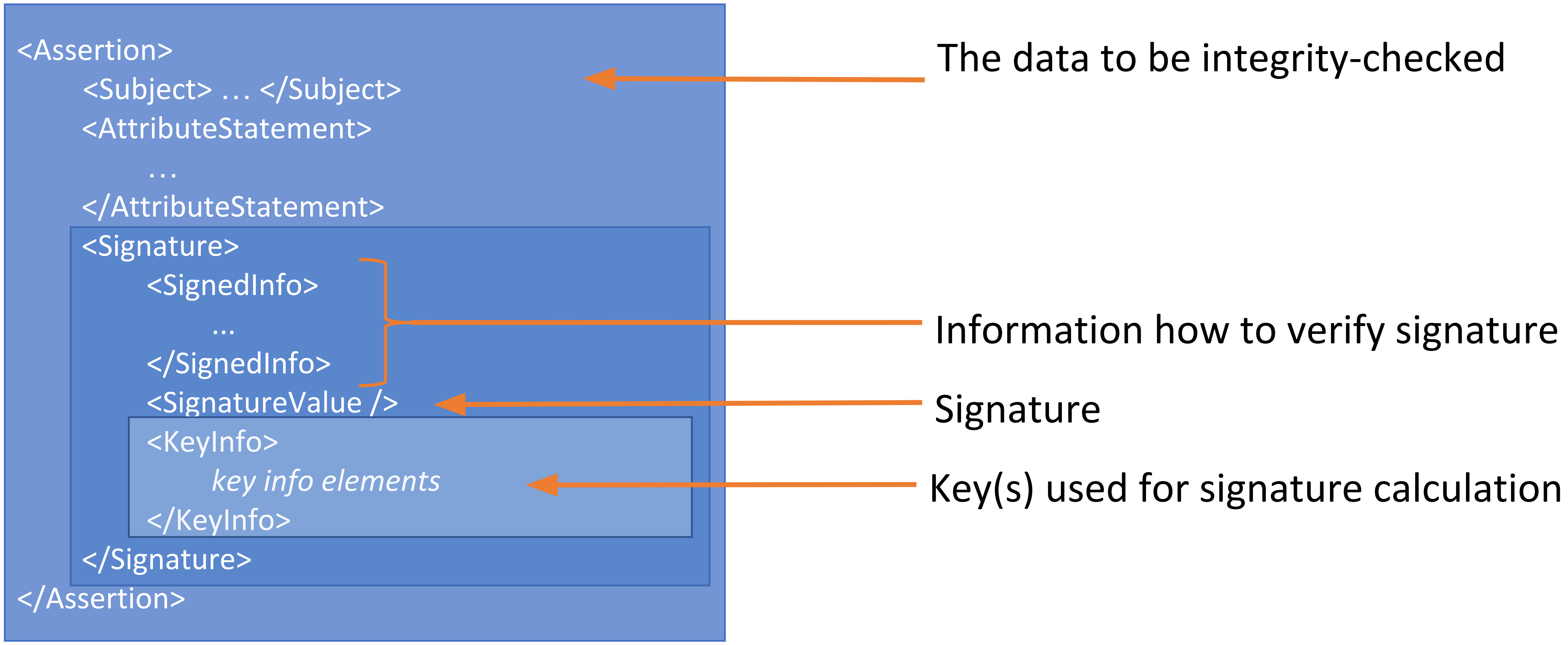
Dupe Key Confusion

CVE-2019-1006

Authentication Tokens - SAML

- The Security Assertion Markup Language, SAML:
  - Popular standard used in single sign-on systems
  - XML-based format
  - Uses XML Signature (aka XMLDSig) standard
- XMLDSig standard (RFC 3275):
  - Used to provide payload security in SAML, SOAP, WS-Security, etc.

Simplified SAML Token



```
<Assertion>
        <Subject> … </Subject>
        <AttributeStatement>
                …
        </AttributeStatement>
        <Signature>
                <SignedInfo>
                        …
                </SignedInfo>
                <SignatureValue />
                <KeyInfo>
                        key info elements
                </KeyInfo>
        </Signature>
</Assertion>
```

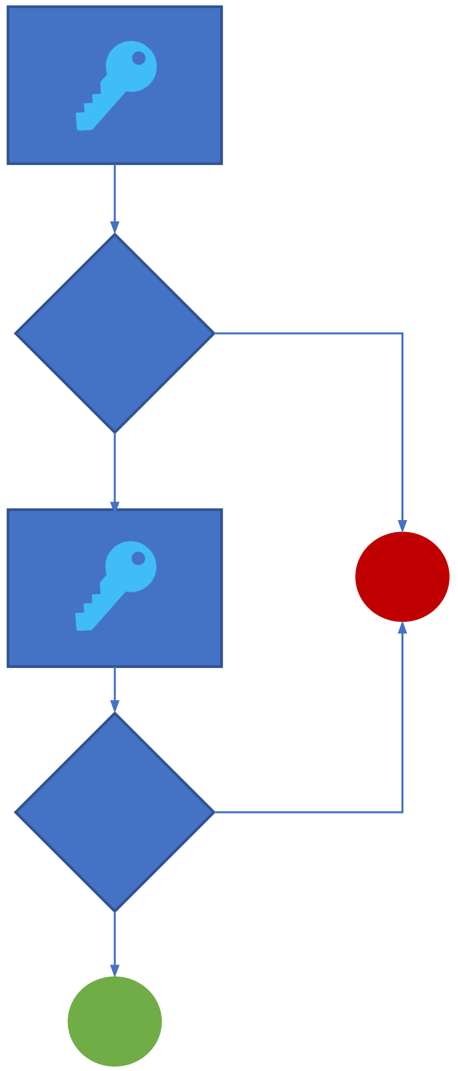
The data to be integrity-checked
Information how to verify signature
Signature
Key(s) used for signature calculation

Previous vulnerabilities in SAML

- XML Signature Wrapping (XSW):
  - Discovered in 2012 by Juraj Somorovsky, Andreas Mayer and others
  - Many implementations in different languages were affected
  - The attacker needs access to a valid token
  - The attacker modifies the contents of the token by injecting malicious data <u>without invalidating the signature</u>
- Attacks with XML comments:
  - Discovered in 2018 by Kelby Ludwig
  - The attacker needs access to a valid token
  - Uses XML comments to modify values <u>without invalidating the signature</u>

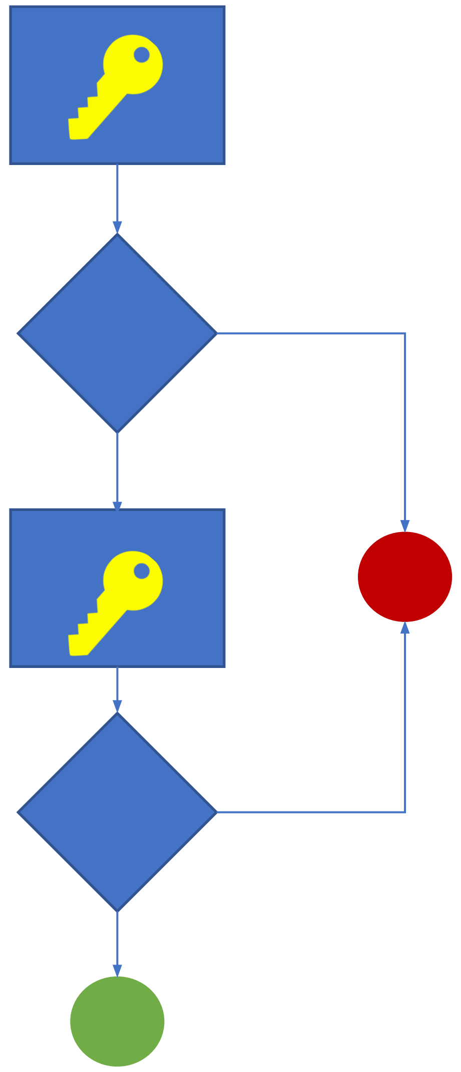
SAML Signature Verification in .NET

1. Resolve the signing key
    - Obtain key from <KeyInfo /> or create it from embedded data
2. Use key to verify signature
3. Identify the signing party
    - Derive `SecurityToken` from <KeyInfo />
4. Authenticate the signing party
    - Verify trust on `SecurityToken`



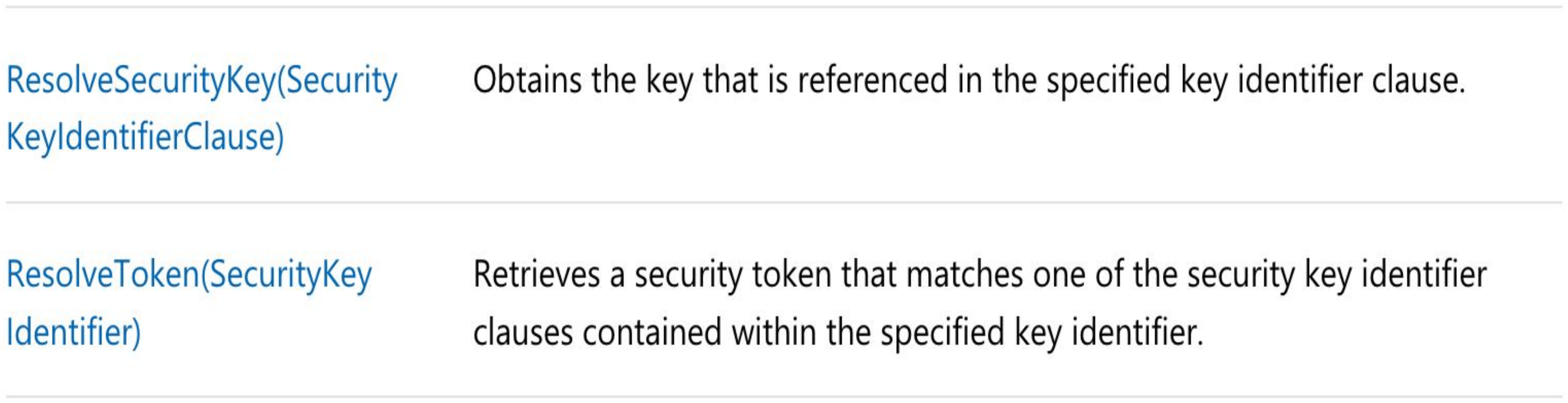
SAML Signature Verification in .NET

1. Resolve the signing key
   - Obtain key from <KeyInfo /> or create it from embedded data
2. Use key to verify signature
3. Identify the signing party
   - Derive `SecurityToken` from <KeyInfo />
4. Authenticate the signing party
   - Verify trust on `SecurityToken`



SecurityTokenResolver

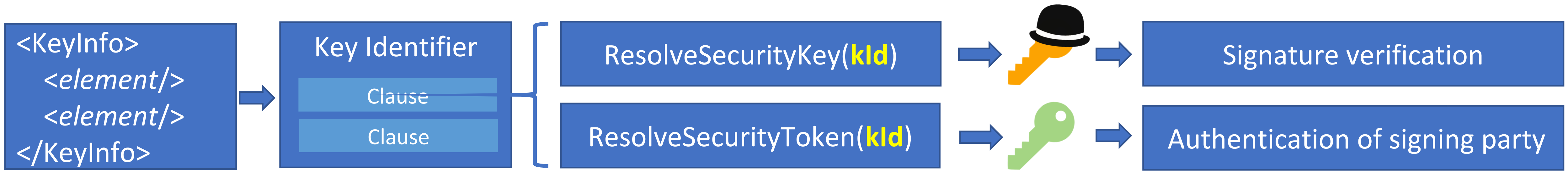
- `System.IdentityModel.Selectors.SecurityTokenResolver`



ResolveSecurityKey(SecurityKeyIdentifierClause)	Obtains the key that is referenced in the specified key identifier clause.
ResolveToken(SecurityKeyIdentifier)	Retrieves a security token that matches one of the security key identifier clauses contained within the specified key identifier.

A tale of two resolvers

- <KeyInfo/> section is processed **twice** by different methods!



Microsoft terminology

- Premise:
  - If we can get each method to return different keys, we may be able to bypass validation

Possible scenarios for different key resolution

1. *Method A* supports a key type that is not supported by *Method B*
2. Both methods support same key types, but in different order
3. Methods check for different subsets of keys within the *<KeyInfo/>* section

Examples of affected frameworks

Windows Communication Foundation (WCF)

- Used in Web Services
- Eg: Exchange server

Windows Identity Foundation (WIF)

- Used in claim-aware applications
- Eg: MVC application authenticating users with ADFS or Azure Active Directory

Windows Identity Foundation (WIF) + Custom configuration

- Uses custom configuration such as a custom resolver or custom certificate store
- Eg: SharePoint

Windows Communication Foundation (WCF)

Windows Communication Foundation (WCF)

- Framework for building service-oriented applications (SOA)
- Interaction between WCF endpoint and client is done using SOAP envelopes (XML documents)
- WCF accepts SAML tokens as Client credentials
- May use Windows Identity Foundation (WIF) or not
- XML Signature also used for proof tokens and other usages

Key & Token Resolution

```
// System.IdentityModel.Tokens.SamlAssertion
SecurityKeyIdentifier keyIdentifier = signedXml.Signature.KeyIdentifier;
this.verificationKey = SamlSerializer.ResolveSecurityKey(keyIdentifier, outOfBandTokenResolver);
if (this.verificationKey == null) throw ...
this.signature = signedXml;
this.signingToken = SamlSerializer.ResolveSecurityToken(keyIdentifier, outOfBandTokenResolver);
```

Same <keyInfo/> block is processed twice

Security Key resolution – Depth First

```
// System.IdentityModel.Tokens.SamlSerializer
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;
            }
        }
    }
...
```

For each <KeyInfo/> element, try ALL resolvers, until one is successful

Security Key resolution – Depth First

Remember, one key at a time!

```
// System.ServiceModel.Security.AggregateSecurityHeaderTokenResolver
bool TryResolveSecurityKeyCore(SecurityKeyIdentifierClause keyIdentifierClause, out SecurityKey key) {
      ...

      resolved = this.tokenResolver.TryResolveSecurityKey(keyIdentifierClause, false, out key);
      if (!resolved)
          resolved = base.TryResolveSecurityKeyCore(keyIdentifierClause, out key);
      if (!resolved)
          resolved = SecurityUtils.TryCreateKeyFromIntrinsicKeyClause(keyIdentifierClause, this, out key);
```

Token resolution – Breadth First

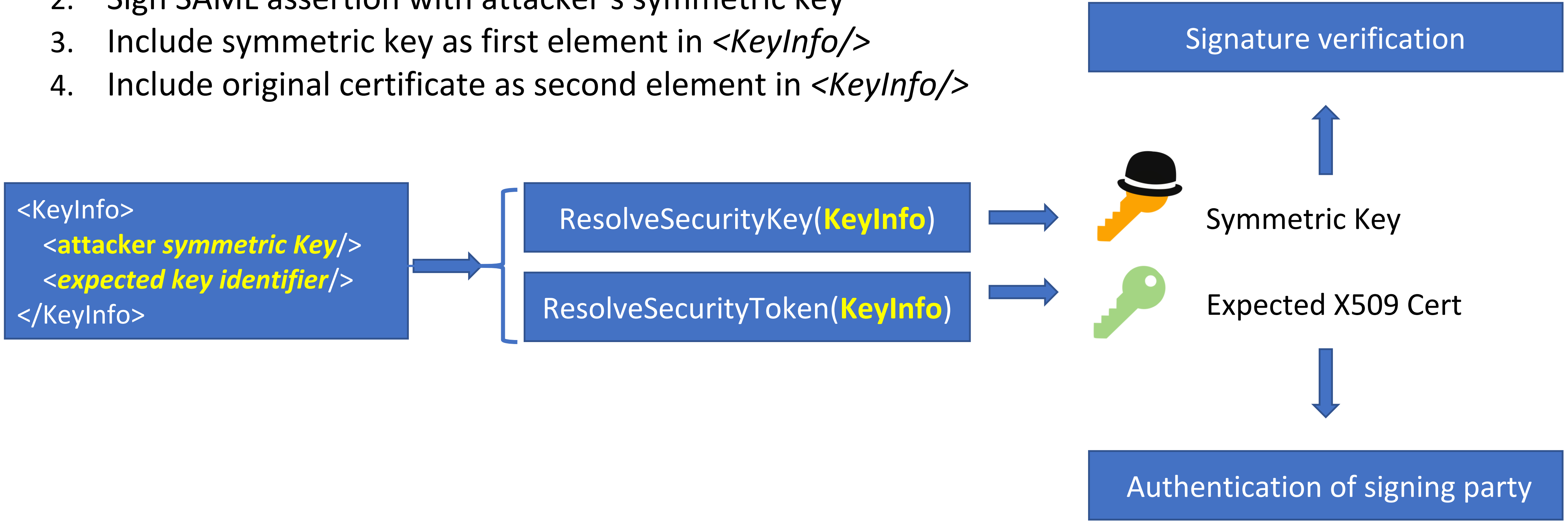
```
// System.ServiceModel.Security.AggregateSecurityHeaderTokenResolver
override bool TryResolveTokenCore(SecurityKeyIdentifier keyIdentifier, out SecurityToken token) {
        bool resolved = false;
        token = null;
        resolved = this.tokenResolver.TryResolveToken(keyIdentifier, false, false, out token);
        if (!resolved) resolved = base.TryResolveTokenCore(keyIdentifier, out token);
        if (!resolved) {
            for (int i = 0; i < keyIdentifier.Count; ++i) {
                if (this.TryResolveTokenFromIntrinsicKeyClause(keyIdentifier[i], out token)) {
                    resolved = true;
                    break;
                }
```

Remember, ALL keys are passed here!

For each token resolver, try ALL <keyInfo/> elements, until one is successful

Dupe Key Confusion

1. Modify token at will or create token from scratch
2. Sign SAML assertion with attacker's symmetric key
3. Include symmetric key as first element in *<KeyInfo/>*
4. Include original certificate as second element in *<KeyInfo/>*



<KeyInfo>
<***attacker symmetric Key***/>
<***expected key identifier***/>
</KeyInfo>
ResolveSecurityKey(**KeyInfo**)
ResolveSecurityToken(**KeyInfo**)
Symmetric Key
Expected X509 Cert
Signature verification
Authentication of signing party

Dupe Key Confusion

```
<ds:KeyInfo>
        <trust:BinarySecret >rV4k60..Oww==</trust:BinarySecret>            Injected Key
        <ds:X509Data>
          <ds:X509Certificate>MIIDBTCCAe2gAw….rzCf6zzzWh</ds:X509Certificate>
        </ds:X509Data>                                                     Original Cert
</ds:KeyInfo>
```

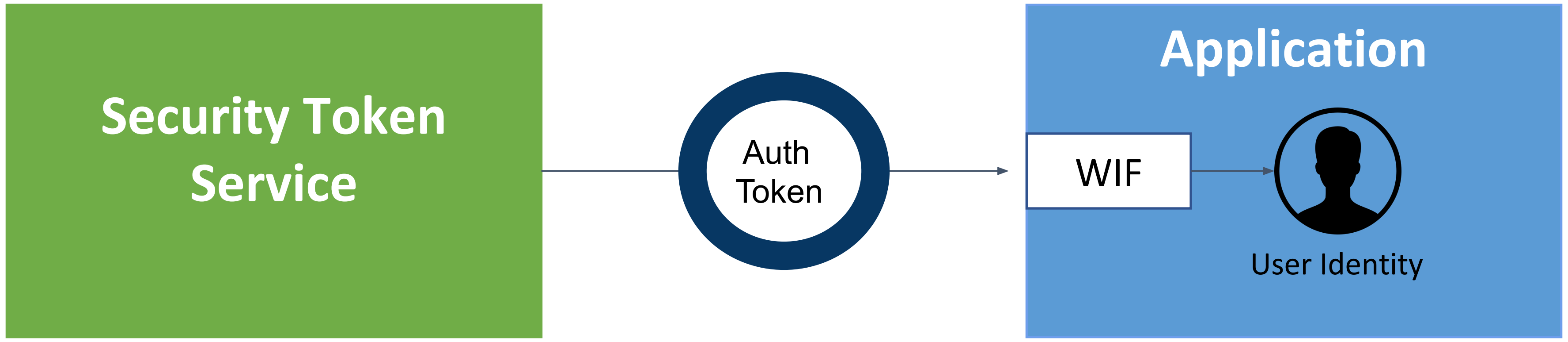
Demo

Exchange Account Takeover

Windows Identity Foundation (WIF)

WIF in a Nutshell

- WIF 4.5 is a framework for building identity-aware applications.
- Applications can use WIF to process tokens issued from STSs (eg: AD FS, Azure AD, ACS, etc.) and make identity-based decisions



Key and Token resolutions

- Key resolution is only attempted with first Key Identifier!

```
if (!tokenResolver.TryResolveSecurityKey(_signedXml.Signature.KeyIdentifier[0], out
key)) {
      ...
}
```

- Security Token resolution is attempted for all Key Identifiers

```
foreach (SecurityKeyIdentifierClause securityKeyIdentifierClause in keyIdentifier) {
      ...
}
```

Key and Token resolutions

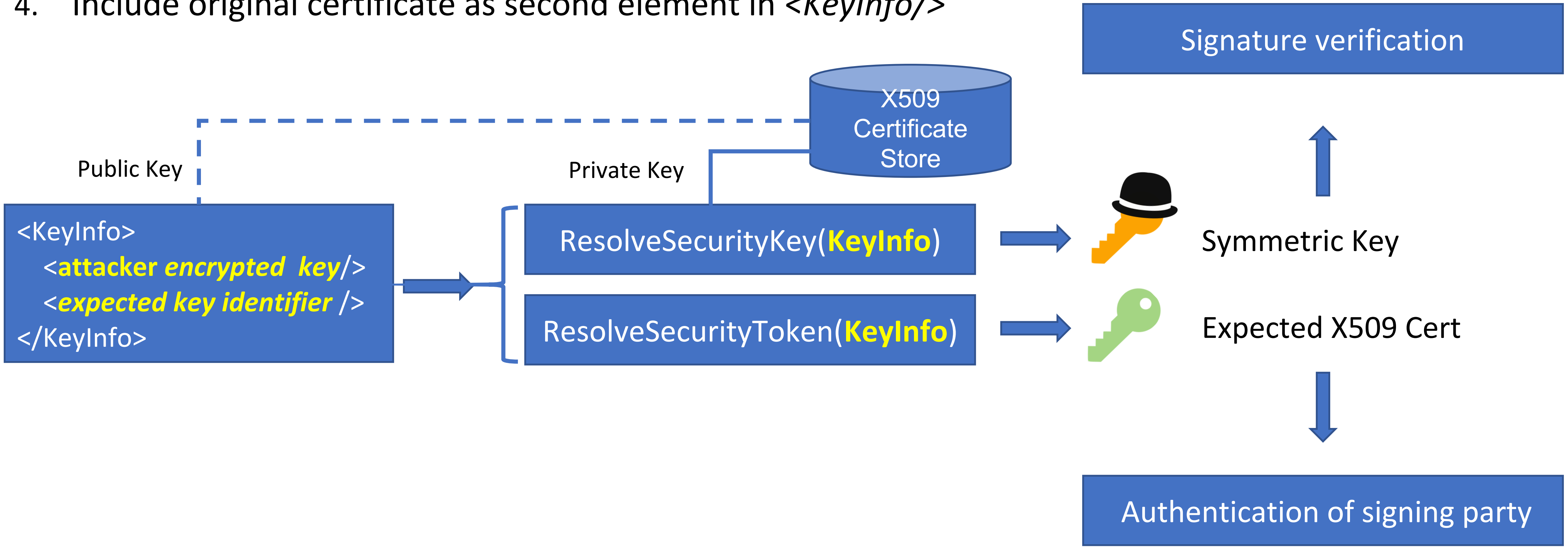
- Uses `System.IdentityModel.Tokens.IssuerTokenResolver`
  - Secure resolver: It handles key and security token resolution in the same way
- Falls back to `X509CertificateStoreTokenResolver` in case of a miss
  - `ResolveSecurityKey()` <u>supports</u> `EncryptedKeyIdentifierClause`
  - `ResolveToken()` <u>only knows about resolving X509 certificates</u>

Attack limitations

- Symmetric key is decrypted using Private key from certificate stored in specific storage
  - By default this storage is **LocalMachine/Trusted People**
- Attacker needs to obtain public key of such certificate
  - Perhaps used for server SSL?

Dupe Key Confusion

1. Re-Sign SAML assertion with attacker's symmetric key
2. Encrypt symmetric key using public key from server certificate
3. Include encrypted symmetric key as first element in *<KeyInfo/>*
4. Include original certificate as second element in *<KeyInfo/>*



Signature verification
X509 Certificate Store
Public Key
Private Key
<KeyInfo>
<***attacker encrypted key***/>
<***expected key identifier*** />
</KeyInfo>
ResolveSecurityKey(**KeyInfo**)
ResolveSecurityToken(**KeyInfo**)
Symmetric Key
Expected X509 Cert
Authentication of signing party

Dupe Key Confusion

```
<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>
```
Injected Key
```
     <ds:X509Data>
          <ds:X509Certificate>MIIDBTCCAe...f6zzzWh</ds:X509Certificate>
     </ds:X509Data>
```
Original Cert
```
</ds:KeyInfo>
```

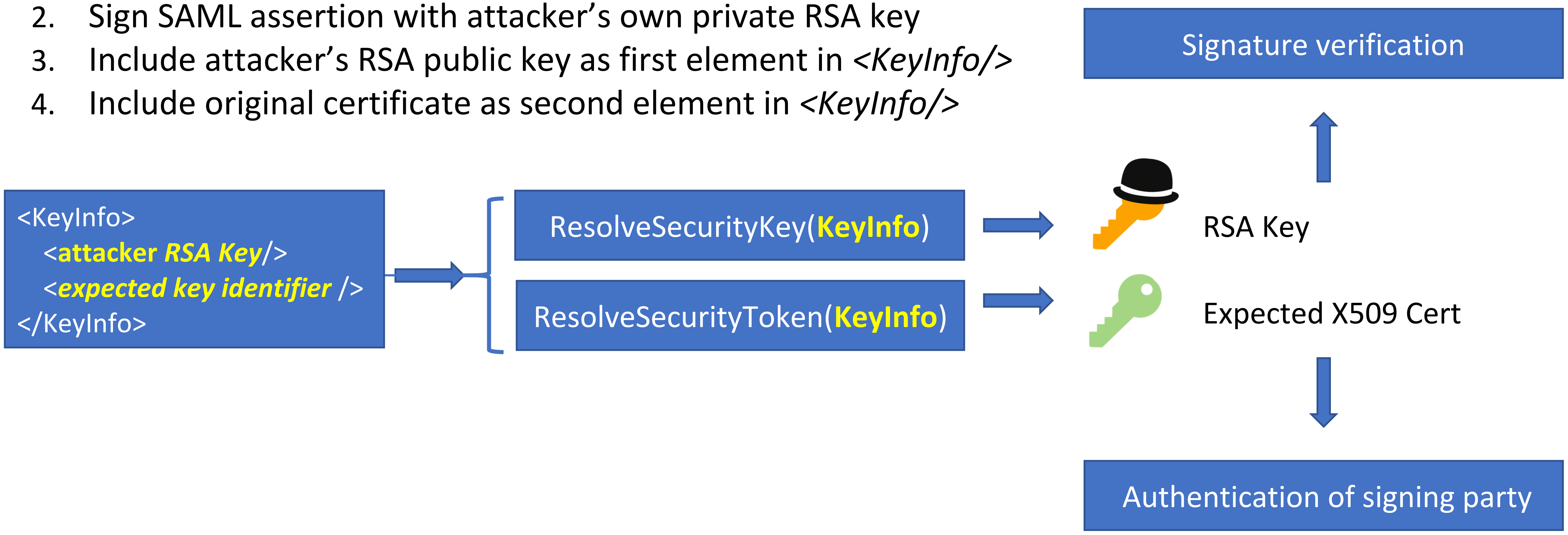
SharePoint Server (WIF)

SharePoint (WIF + Custom resolver)

- SharePoint uses WIF to process tokens and create user identities
- However, it uses a custom security token resolver:
  - `Microsoft.SharePoint.IdentityModel.SPIssuerTokenResolver`
- Key resolution <u>supports</u> Intrinsic keys (eg: RSA Key, BinarySecret, …)
- Token resolution <u>does not know</u> how to resolve Intrinsic keys

Dupe Key Confusion

1. Modify token at will or create token from scratch
2. Sign SAML assertion with attacker's own private RSA key
3. Include attacker's RSA public key as first element in *<KeyInfo/>*
4. Include original certificate as second element in *<KeyInfo/>*

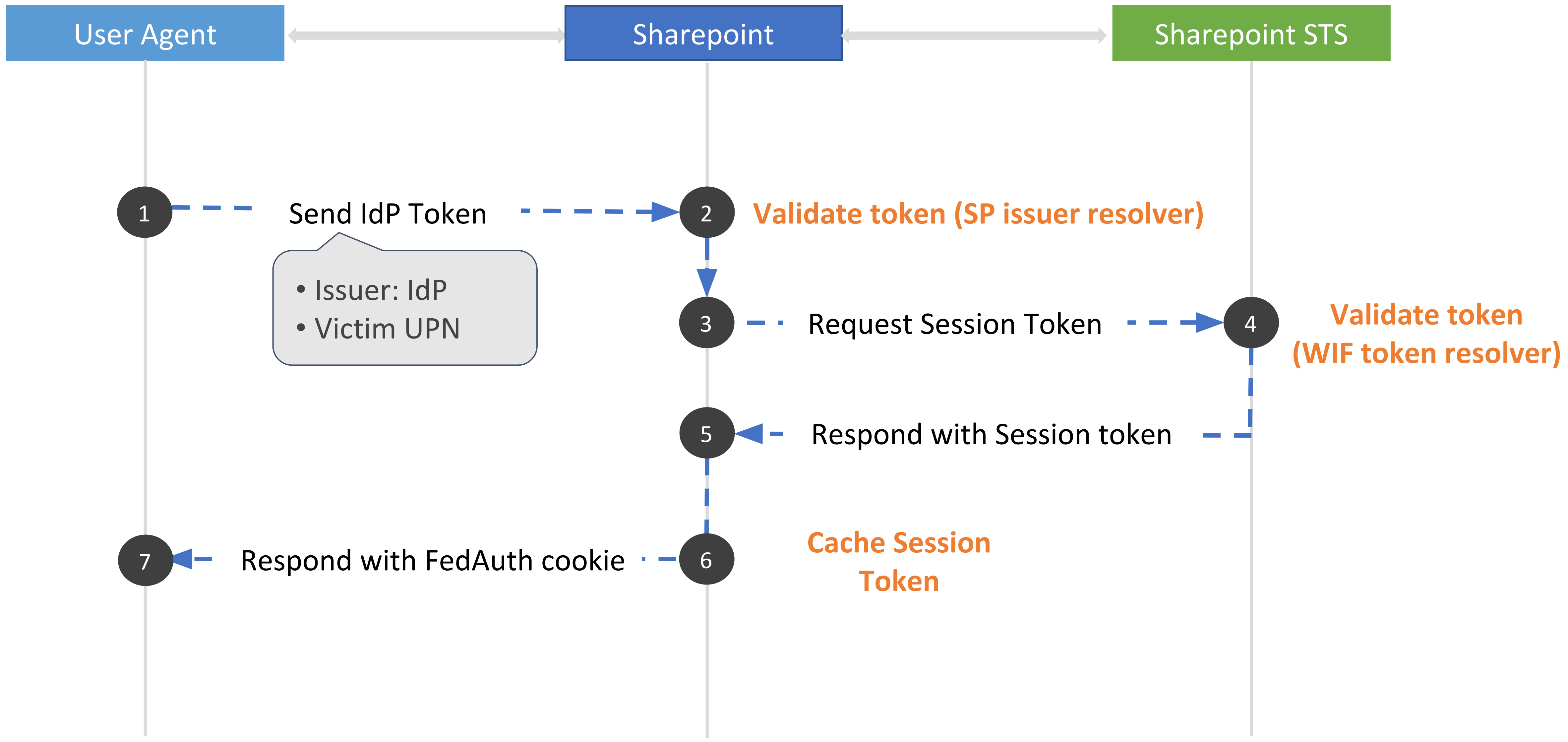


<KeyInfo>
<**attacker *RSA Key***/>
<***expected key identifier*** />
</KeyInfo>
ResolveSecurityKey(**KeyInfo**)
ResolveSecurityToken(**KeyInfo**)
RSA Key
Expected X509 Cert
Signature verification
Authentication of signing party

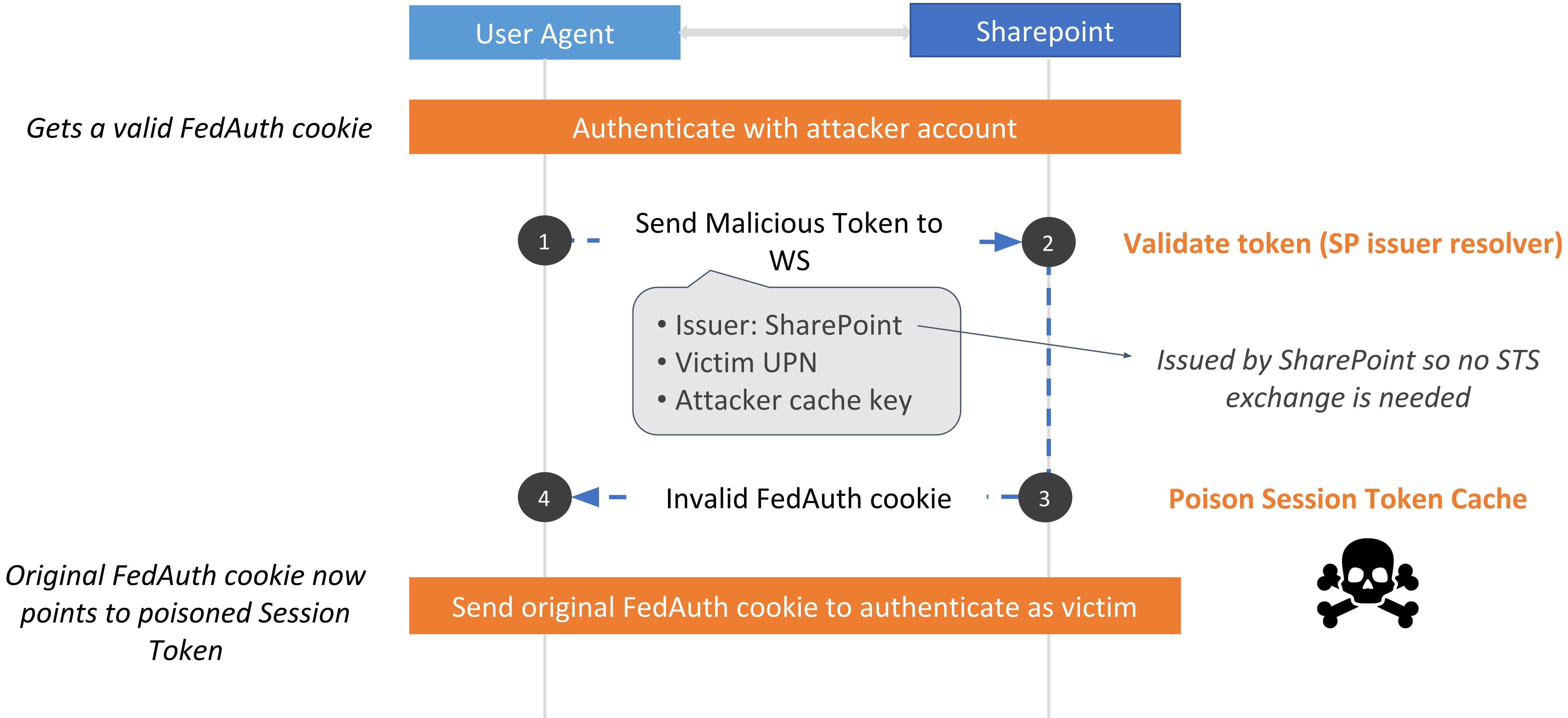
Dupe Key Confusion

```
<ds:KeyInfo>
    <ds:KeyValue>                                                     Injected Key
        <ds:RSAKeyValue>
            <ds:Modulus>irXhaxafoUZ...77kw==</ds:Modulus>
            <ds:Exponent>AQAB</ds:Exponent>
        </ds:RSAKeyValue>
    </ds:KeyValue>
    <ds:X509Data>
        <ds:X509Certificate>MIIDBTCCAe2...zzWh</ds:X509Certificate>
    </ds:X509Data>                                                    Original Cert
</ds:KeyInfo>
```

SharePoint Authentication Flow


User Agent
Sharepoint
Sharepoint STS
1
Send IdP Token
2
Validate token (SP issuer resolver)
• Issuer: IdP
• Victim UPN
3
Request Session Token
4
Validate token (WIF token resolver)
5
Respond with Session token
6
Cache Session Token
7
Respond with FedAuth cookie

SharePoint Attack Flow

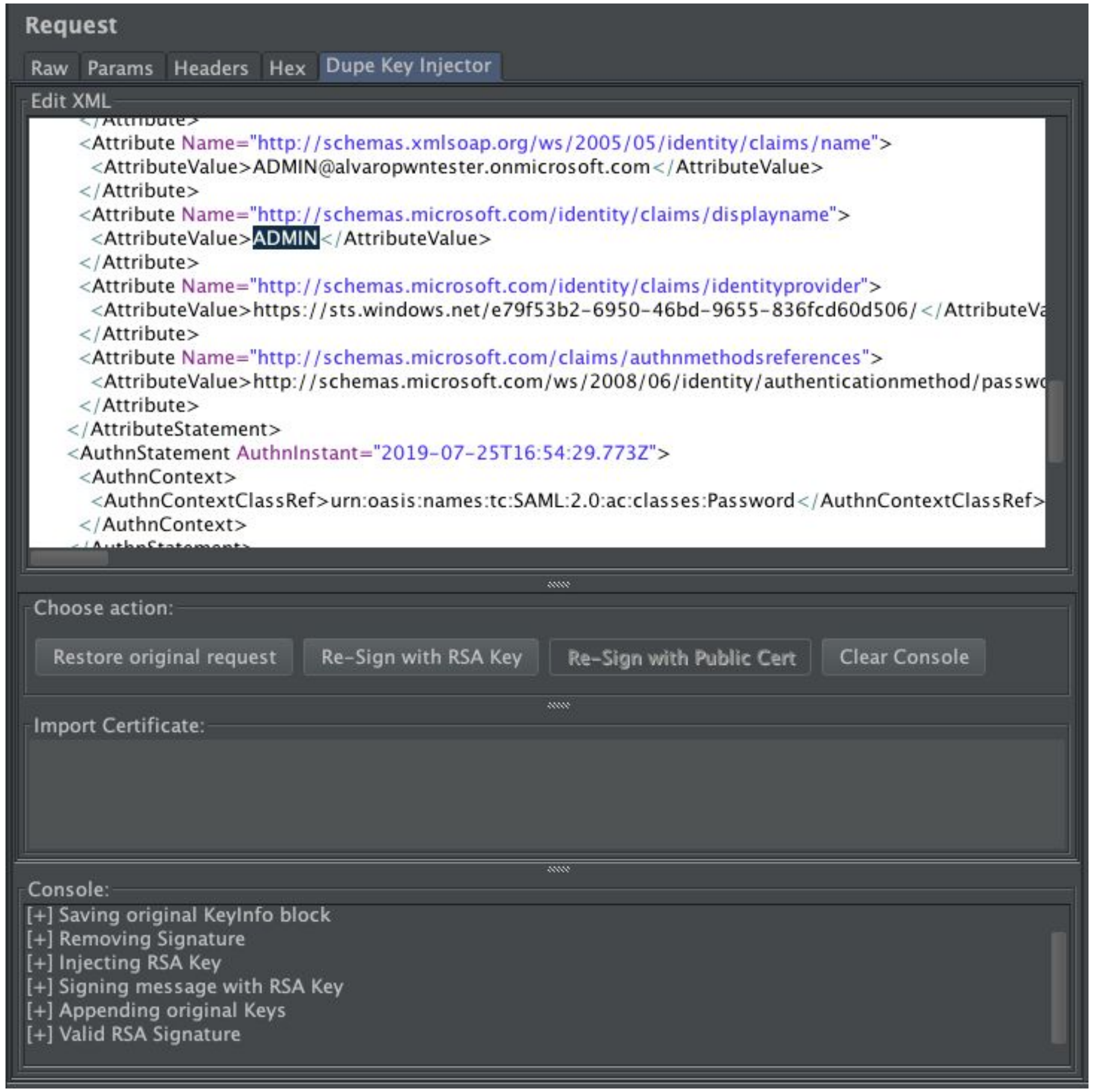

User Agent
Sharepoint
Gets a valid FedAuth cookie
Authenticate with attacker account
1
Send Malicious Token to WS
2
Validate token (SP issuer resolver)
• Issuer: SharePoint
• Victim UPN
• Attacker cache key
Issued by SharePoint so no STS exchange is needed
4
Invalid FedAuth cookie
3
Poison Session Token Cache
Original FedAuth cookie now points to poisoned Session Token
Send original FedAuth cookie to authenticate as victim

Demo

Privilege escalation on SharePoint

Burp Plugin

https://github.com/pwntester/DupeKeyInjector



```
  </Attribute>
  <Attribute Name="http://schemas.xmlsoap.org/ws/2005/05/identity/claims/name">
    <AttributeValue>ADMIN@alvaropwntester.onmicrosoft.com</AttributeValue>
  </Attribute>
  <Attribute Name="http://schemas.microsoft.com/identity/claims/displayname">
    <AttributeValue>ADMIN</AttributeValue>
  </Attribute>
  <Attribute Name="http://schemas.microsoft.com/identity/claims/identityprovider">
    <AttributeValue>https://sts.windows.net/e79f53b2-6950-46bd-9655-836fcd60d506/</AttributeVa
  </Attribute>
  <Attribute Name="http://schemas.microsoft.com/claims/authnmethodsreferences">
    <AttributeValue>http://schemas.microsoft.com/ws/2008/06/identity/authenticationmethod/passwo
  </Attribute>
</AttributeStatement>
<AuthnStatement AuthnInstant="2019-07-25T16:54:29.773Z">
  <AuthnContext>
    <AuthnContextClassRef>urn:oasis:names:tc:SAML:2.0:ac:classes:Password</AuthnContextClassRef>
  </AuthnContext>
```

```
[+] Saving original KeyInfo block
[+] Removing Signature
[+] Injecting RSA Key
[+] Signing message with RSA Key
[+] Appending original Keys
[+] Valid RSA Signature
```

Conclusions & Takeaways

Conclusions

- Even if protocols are considered secure, the devil is in the implementations
- Processing same data with inconsistent code may lead to vulnerabilities
- Here be dragons:
    - Research focused on .NET, similar flaws can exist in other languages
    - Even in .NET, XML Signature is used in other potentially insecure places
- Patch ASAP :)

@Pwntester
@OlekMirosh

Questions?