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

1. Access protected resource
2. Redirect to SSO service
3. Login into SSO service
4. Respond with Auth token
5. Forward Auth token
6. Redirect to resource
7. Access resource

Service Provider → User Agent → Identity Provider
Delegated Authentication

1. Access protected resource
2. Redirect to SSO service
3. Login into SSO service
4. Respond with Auth token
5. Forward Auth token
6. Redirect to resource
7. Access resource
8. Resource

Issuer
Audience
Expire Date
Claims
Signature
Delegated Authentication

1. Access protected resource
2. Redirect to SSO service
3. Login into SSO service
4. Respond with Auth token
5. Forward Auth token
6. Redirect to resource
7. Access resource
8. Resource

Issuer
Audience
Expire Date
Claims
Signature
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

Decoded
EDIT THE PAYLOAD AND SECRET

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIvLCJlbWFpbCI6InBheWQgQ2VydGlmaWNhdGVkIiwibmFtZSI6ImlubSIsImF1ZCI6IkpvaG4gRG9lIiwidHlwZSI6IkpvaG4iLCJwaHNpZ25hdGlvbiI6Ik9yZyIsInN1YiI6IjEzNzE2M6h8IiwiY29yZCI6IjEzNzE2M6h8IiwiYXV0aG9yaXR5IjoiMTk3LjEiLCJpYXQiOjEzNzE2M6h8Iiwic3VwZGF0aW9uIjoie1wifQ.
```

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

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

Source: http://jwt.io
System.IdentityModel.Tokens.Jwt library

```csharp
// 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);
    ...
}
```
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;
    }
    ...
    RuntimeMethod runtimeType = type as RuntimeMethod;
    ...
    MethodBase[] array = runtimeType.GetConstructors(BindingFlags.Instance | BindingFlags.Public | BindingFlags.CreateInstance);
    ...
    object obj;
    ...
    object result = runtimeConstructorInfo.Invoke(BindingFlags.Instance | BindingFlags.Public | BindingFlags.CreateInstance, Type.DefaultBinder, args, null);
Similar code for SAML

```xml
<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>
```

```csharp
// 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.:

    ```csharp
    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 😞

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

Second payload: ../../../../../../tmp/malicious

```csharp
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);
    }
    ...
```
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

```xml
<Assertion>
  <Subject> … </Subject>
  <AttributeStatement>
    …
  </AttributeStatement>
</Assertion>

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

- 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 **without invalidating the signature**

• Attacks with XML comments:
  • Discovered in 2018 by Kelby Ludwig
  • The attacker needs access to a valid token
  • Uses XML comments to modify values **without invalidating the signature**
SAML Signature Verification in .NET

1. Resolve the signing key
   - Obtain SecurityKey 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 **SecurityKey** 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**


<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ResolveSecurityKey(SecurityKeyIdentifierClause)</code></td>
<td>Obtains the key that is referenced in the specified key identifier clause.</td>
</tr>
<tr>
<td><code>ResolveToken(SecurityKeyIdentifier)</code></td>
<td>Retrieves a security token that matches one of the security key identifier clauses contained within the specified key identifier.</td>
</tr>
</tbody>
</table>
A tale of two resolvers

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

![Diagram showing the process of processing `<KeyInfo/>` section](image)

**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
// 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

```csharp
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

```csharp
override bool TryResolveTokenCore(SecurityKeyIdentifier keyIdentifier, out SecurityToken token) {
    bool resolved = false;
    token = null;
    resolved = this.tokenResolver.TryResolveToken(keyIdentifier, false, 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;
            }
        }
    }
    return resolved;
}
```

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/>

```xml
<KeyInfo>
  <attacker symmetric Key/>
  <expected key identifier/>
</KeyInfo>
```

- ResolveSecurityKey(KeyInfo)
- ResolveSecurityToken(KeyInfo)

---

Signature verification

- Symmetric Key
- Expected X509 Cert

Authentication of signing party
Dupe Key Confusion

<ds:KeyInfo>
  <trust:BinarySecret>rV4k60..Oww==</trust:BinarySecret>
  <ds:X509Data>
    <ds:X509Certificate>MIIDBTCCAe2gAw....rzCf6zzzWh</ds:X509Certificate>
  </ds:X509Data>
</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!

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

- Security Token resolution is attempted for all Key Identifiers

```csharp
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()` **supports** `EncryptedKeyIdentifierClause`
  - `ResolveToken()` **only knows about resolving X509 certificates**
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
- 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:X509Data>
    </ds:KeyInfo>
    <xenc:CipherData>
      <xenc:CipherValue>e++....</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedKey>
</ds:KeyInfo>

<ds:X509Data>
  <ds:X509Certificate>MIIDBTCC...f6zzzWh</ds:X509Certificate>
</ds:X509Data>
</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 supports Intrinsic keys (eg: RSA Key, BinarySecret, …)

• Token resolution does not know 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/>

```xml
<KeyInfo>
  <attacker RSA Key/>
  <expected key identifier />
</KeyInfo>
```

```
ResolveSecurityKey(KeyInfo)
ResolveSecurityToken(KeyInfo)
```

Signature verification

- RSA Key
- Expected X509 Cert

Authentication of signing party
Dupe Key Confusion

<ds:KeyValue>
    <ds:RSAKeyValue>
        <ds:Modulus>irXhxafoUZ...77kw==</ds:Modulus>
        <ds:Exponent>AQAB</ds:Exponent>
    </ds:RSAKeyValue>
</ds:KeyValue>
<ds:X509Data>
    <ds:X509Certificate>MIIIDBTCCAe2...zzWh</ds:X509Certificate>
</ds:X509Data>
</ds:KeyInfo>
SharePoint Authentication Flow

1. Send IdP Token
   - Issuer: IdP
   - UPN claim: user

2. Validate token (SP issuer resolver)

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

1. Send Malicious Token to WS
   - Issuer: SharePoint
   - UPN claim: victim
   - Cache key: attacker
2. Validate token (SP issuer resolver)
3. Poison Session Token Cache
4. Invalid FedAuth cookie

User Agent

Authenticate with attacker account

Gets a valid FedAuth cookie

Original FedAuth cookie now points to poisoned Session Token

Send original FedAuth cookie to authenticate as victim
Demo
Privilege escalation on SharePoint
Burp Plugin
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?