NOT FOR RELEASE – Under embargo until Tuesday, Nov. 21, at 8 a.m. EST

Golden SAML: Newly Discovered Attack Technique Forges Authentication to Cloud Apps

By Shaked Reiner, CyberArk Labs Security Researcher

Abstract

In this blog post, we introduce a new attack vector discovered by CyberArk Labs and dubbed “golden SAML.” The vector enables an attacker to create a golden SAML, which is basically a forged SAML “authentication object,” and authenticate across every service that uses SAML 2.0 protocol as an SSO mechanism.

In a golden SAML attack, attackers can gain access to any application that supports SAML authentication (e.g. Azure, AWS, vSphere, etc.) with any privileges they desire and be any user on the targeted application (even one that is non-existent in the application in some cases).

We are releasing a new tool that implements this attack – *shimit*.

In a time when more and more enterprise infrastructure is ported to the cloud, the Active Directory (AD) is no longer the highest authority for authenticating and authorizing users. AD can now be part of something bigger – a federation.

A federation enables trust between different environments otherwise not related, like Microsoft AD, Azure, AWS and many others. This trust allows a user in an AD, for example, to be able to enjoy SSO benefits to all the trusted environments in such federation. Talking about a federation, an attacker will no longer suffice in dominating the domain controller of his victim.

The golden SAML name may remind you of another notorious attack known as golden ticket*,* which was introduced by Benjamin Delpy who is known for his famous attack tool called Mimikatz. The name resemblance is intended, since the attack nature is rather similar. Golden SAML introduces to a federation the advantages that golden ticket offers in a Kerberos environment – from gaining any type of access to stealthily maintaining persistency.

SAML Explained

For those of you who aren’t familiar with the SAML 2.0 protocol, we’ll take a minute to explain how it works.

The SAML protocol, or **Security Assertion Markup Language,** is an open standard for exchanging authentication and authorization data between parties, in particular, between an identity provider and a service provider. Beyond what its name suggests, SAML is each of the following:

- An XML-based markup language (for assertions, etc.)
- A set of XML-based protocol messages
- A set of protocol message bindings
- A set of profiles (utilizing all of the above)

The single most important use case that SAML addresses is web browser single sign-on (SSO). [Wikipedia]

Let’s take a look at *figure 1* in order to understand how this protocol works.

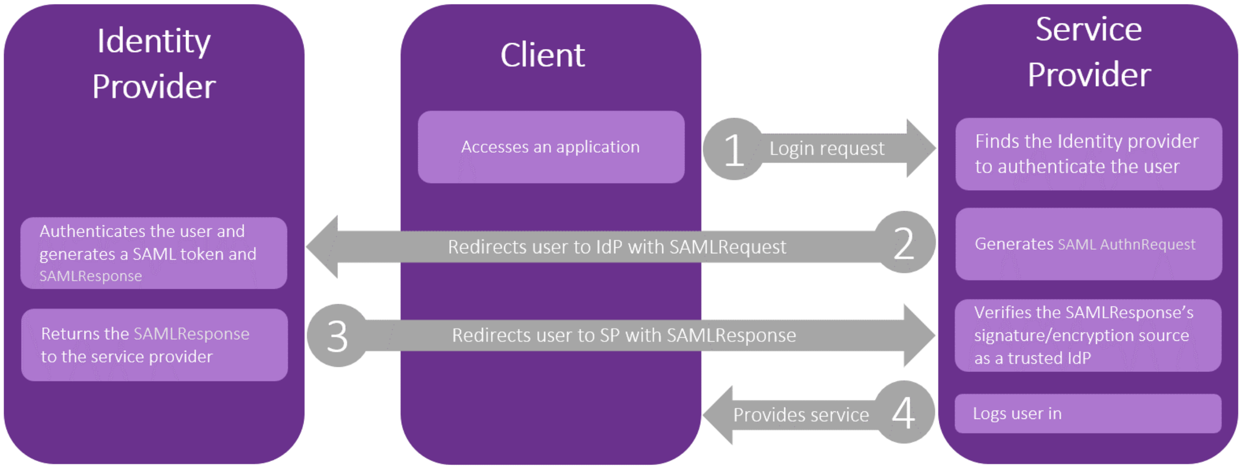

Identity Provider
Client
Service Provider
Accesses an application
1 Login request
Finds the Identity provider to authenticate the user
Authenticates the user and generates a SAML token and SAMLResponse
Redirects user to IdP with SAMLRequest 2
Generates SAML AuthnRequest
Returns the SAMLResponse to the service provider
3 Redirects user to SP with SAMLResponse
Verifies the SAMLResponse’s signature/encryption source as a trusted IdP
Provides service 4
Logs user in

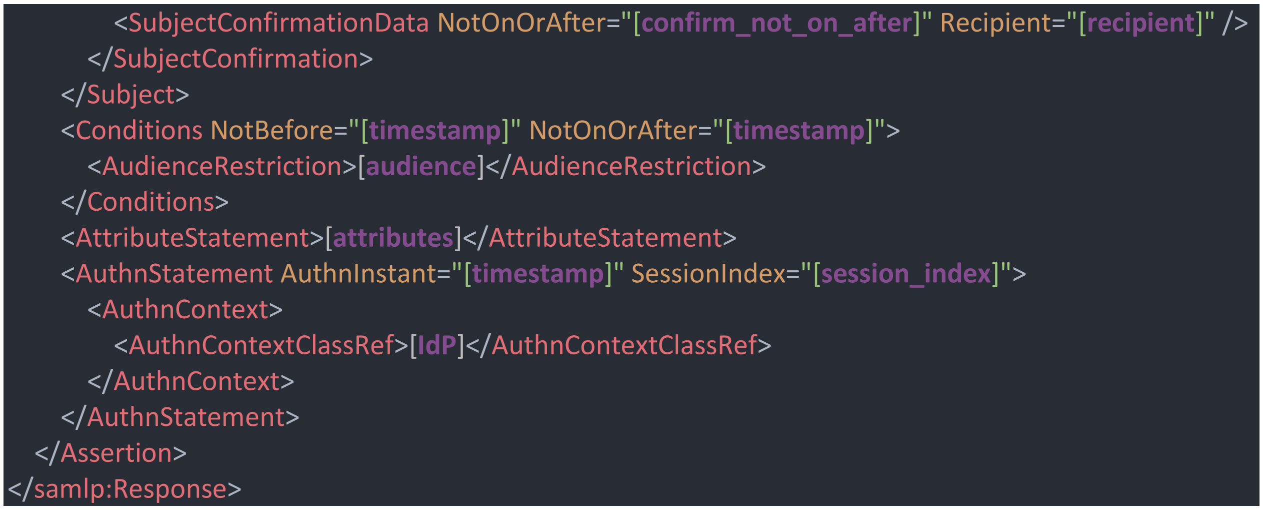
Figure 1- SAML Authentication

1. First the user tries to access an application (also known as the SP i.e. Service Provider), that might be an AWS console, vSphere web client, etc. Depending on the implementation, the client may go directly to the IdP first, and skip the first step in this diagram.
2. The application then detects the IdP (i.e. Identity Provider, could be AD FS, Okta, etc.) to authenticate the user, generates a SAML AuthnRequest and redirects the client to the IdP.
3. The IdP authenticates the user, creates a SAMLResponse and posts it to the SP via the user.
4. SP checks the SAMLResponse and logs the user in. The SP must have a trust relationship with the IdP. The user can now use the service.

SAML Response Structure

Talking about a golden SAML attack, the part that interests us the most is #3, since this is the part we are going to replicate as an attacker performing this kind of attack. To be able to perform this correctly, let’s have a look at the request that is sent in this part – SAMLResponse. The SAMLResponse object is what the IdP sends to the SP, and this is actually the data that makes the SP identify and authenticate the user (similar to a *TGT* generated by a *KDC* in Kerberos). The general structure of a SAMLResponse in SAML 2.0 is as follows (written in purple are all the dynamic parameters of the structure):

```
<samlp:Response ID="[id]" Version="2.0" IssueInstant="[timestamp]"
Destination="[SP]" Consent="urn:oasis:names:tc:SAML:2.0:consent:[consent]"
xmlns:samlp="urn:oasis:names:tc:SAML:2.0:protocol">
  <Issuer xmlns="urn:oasis:names:tc:SAML:2.0:assertion">[issuer]</Issuer>
  <samlp:Status>
    <samlp:StatusCode Value="urn:oasis:names:tc:SAML:2.0:status:[status]" />
  </samlp:Status>
  <Assertion ID="[id]" IssueInstant="[timestamp]" Version="2.0"
xmlns="urn:oasis:names:tc:SAML:2.0:assertion">
    <Issuer>[IdP]</Issuer>
    <Subject>
      <NameID Format="urn:oasis:names:tc:SAML:2.0:nameid-format:persistent">[user]</NameID>
      <SubjectConfirmation Method="urn:oasis:names:tc:SAML:2.0:cm:bearer">
```



```
        <SubjectConfirmationData NotOnOrAfter="[confirm_not_on_after]" Recipient="[recipient]" />
      </SubjectConfirmation>
    </Subject>
    <Conditions NotBefore="[timestamp]" NotOnOrAfter="[timestamp]">
      <AudienceRestriction>[audience]</AudienceRestriction>
    </Conditions>
    <AttributeStatement>[attributes]</AttributeStatement>
    <AuthnStatement AuthnInstant="[timestamp]" SessionIndex="[session_index]">
      <AuthnContext>
        <AuthnContextClassRef>[IdP]</AuthnContextClassRef>
      </AuthnContext>
    </AuthnStatement>
  </Assertion>
</samlp:Response>
```

Depending on the specific IdP implementation, the response assertion may be either signed or encrypted by the private key of the IdP. This way, the SP can verify that the SAMLResponse was indeed created by the trusted IdP.

Similar to a golden ticket attack, if we have the key that signs the object which holds the user's identity and permissions (*KRBTGT* for golden ticket and token-signing private key for golden SAML), we can then forge such an "authentication object" (TGT or SAMLResponse) and impersonate any user to gain unauthorized access to the SP. Roger Grimes defined a golden ticket attack back in 2014 not as a Kerberos tickets forging attack, but as a Kerberos Key Distribution Center (KDC) forging attack. Likewise, a golden SAML attack can also be defined as an IdP forging attack.

In this attack, an attacker can control every aspect of the SAMLResponse object (e.g. username, permission set, validity period and more). In addition, golden SAMLs have the following advantages:

- They can be **generated** from practically **anywhere**. You don't need to be a part of a domain, federation of any other environment you're dealing with
- They are effective even when **2FA** is enabled
- The token-signing **private key** is **not renewed** automatically
- Changing a user's password won't affect the generated SAML

AWS + AD FS + Golden SAML = ♥ (case study)

Let's say you are an attacker. You have compromised your target's domain, and you are now trying to figure out how to continue your hunt for the final goal. What's next? One option that is now available for you is using a golden SAML to further compromise assets of your target.

Active Directory Federation Services (**AD FS**) is a Microsoft standards-based domain service that allows the secure sharing of identity information between trusted business partners (federation). It is basically a service in a domain that provides domain user identities to other service providers within a federation.

Assuming AWS trusts the domain which you've compromised (in a federation), you can then take advantage of this attack and practically gain any permissions in the cloud environment. To perform this attack, you'll need the private key that signs the SAML objects (similarly to the need for the KRBTGT in a golden ticket). For this private key, you don't need a domain admin access, you'll only need the AD FS user account.

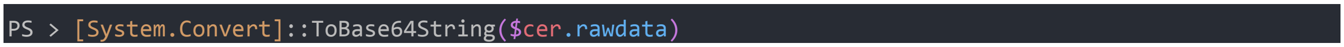
Here’s a list of the requirements for performing a golden SAML attack:

- **Token-signing private key**
- **IdP public certificate**
- **IdP name**
- **Role name (role to assume)**
- Domain\username
- Role session name in AWS
- Amazon account ID

The mandatory requirements are highlighted in purple. For the other non-mandatory fields, you can enter whatever you like.

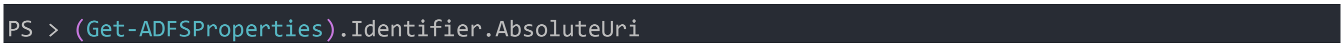
How do you get these requirements? For the private key you’ll need access to the AD FS account, and from its personal store you’ll need to export the private key (export can be done with tools like mimikatz). For the other requirements you can import the powershell snapin Microsoft.Adfs.Powershell and use it as follows (you have to be running as the ADFS user):

ADFS Public Certificate



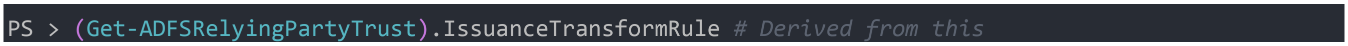
```
PS > [System.Convert]::ToBase64String($cer.rawdata)
```

IdP Name



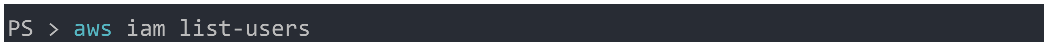
```
PS > (Get-ADFSProperties).Identifier.AbsoluteUri
```

Role Name

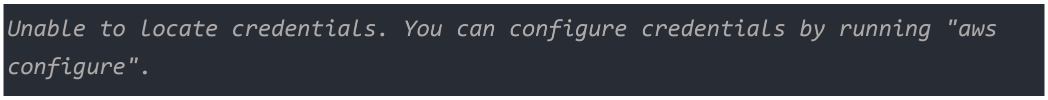


```
PS > (Get-ADFSRelyingPartyTrust).IssuanceTransformRule # Derived from this
```

Once we have what we need, we can jump straight into the attack. First, let’s check if we have any valid AWS credentials on our machine.

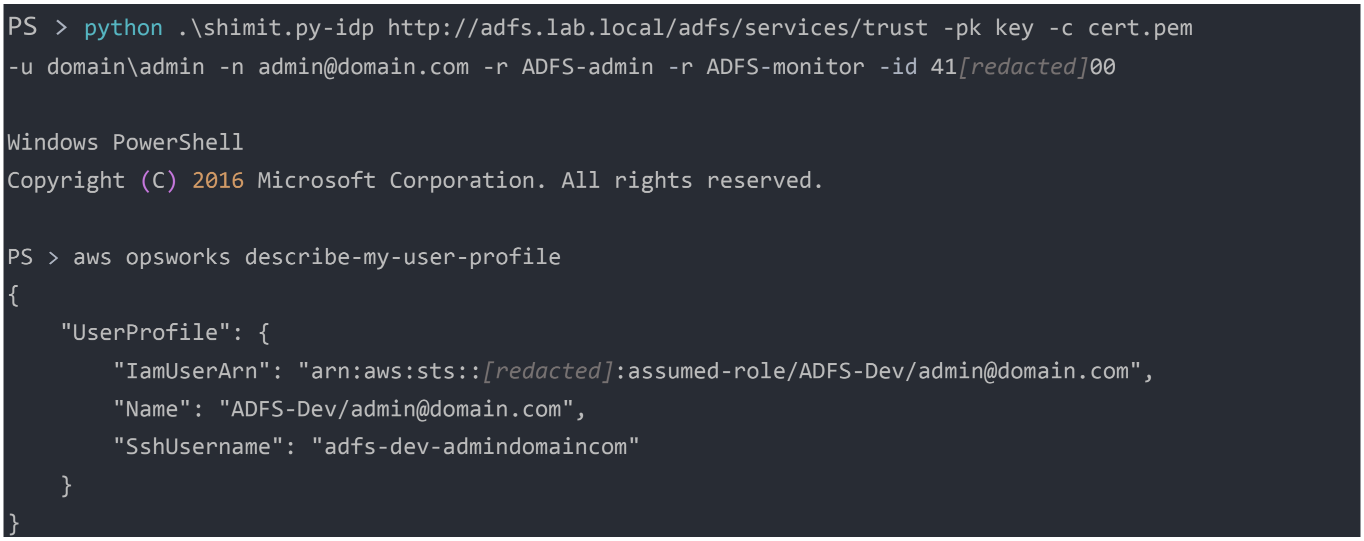


```
PS > aws iam list-users
```



```
Unable to locate credentials. You can configure credentials by running "aws
configure".
```

Unsurprisingly, we have no credentials, but that’s about to change. Now, let’s use shimit to generate and sign a SAMLResponse.



```
PS > python .\shimit.py-idp http://adfs.lab.local/adfs/services/trust -pk key -c cert.pem
-u domain\admin -n admin@domain.com -r ADFS-admin -r ADFS-monitor -id 41[redacted]00

Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS > aws opsworks describe-my-user-profile
{
    "UserProfile": {
        "IamUserArn": "arn:aws:sts::[redacted]:assumed-role/ADFS-Dev/admin@domain.com",
        "Name": "ADFS-Dev/admin@domain.com",
        "SshUsername": "adfs-dev-admindomaincom"
    }
}
```

The operation of the tool is as follows:

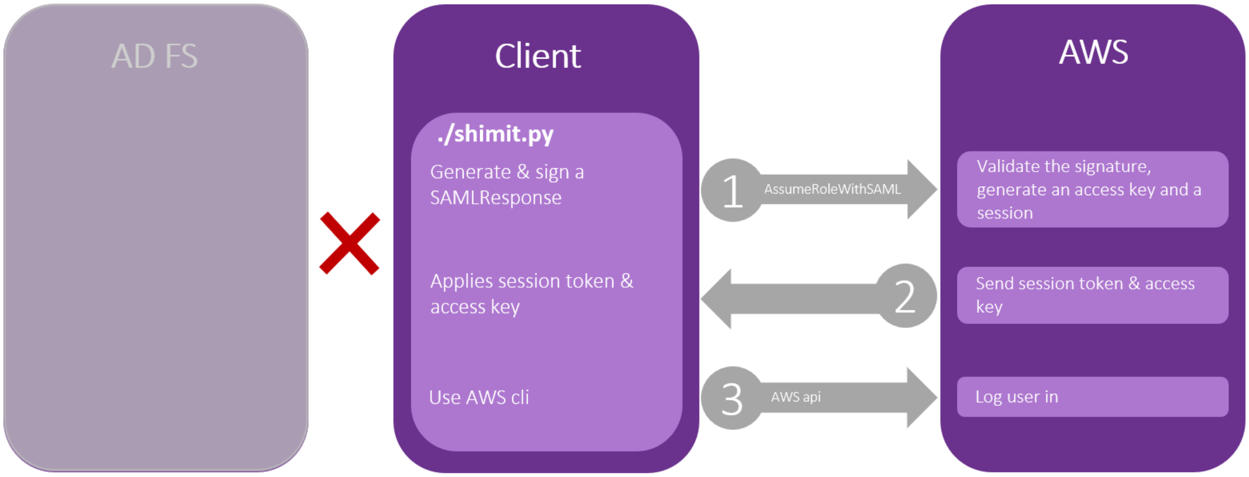

AD FS
Client
./shimit.py
Generate & sign a SAMLResponse
Applies session token & access key
Use AWS cli
1
AssumeRoleWithSAML
2
3
AWS api
AWS
Validate the signature, generate an access key and a session
Send session token & access key
Log user in

Figure 2– Golden SAML with shimit.py

1.
   A. Generate an assertion matching the parameters provided by the user. In this example, we provided the username, Amazon account ID and the desired roles (the first one will be assumed).
   B. Sign the assertion with the private key file, also specified by the user.
   C. Open a connection to the SP, then calling a specific AWS API AssumeRoleWithSAML.
2. Get an access key and a session token from AWS STS (the service that supplies temporary credentials for federated users).

3. Apply this session to the command line environment (using aws-cli environment variables) for the user to use with AWS cli.

Performing a golden SAML attack in this environment has a limitation. Even though we can generate a SAMLResponse that will be valid for any time period we choose (using the --*SamlValidity* flag), AWS specifically checks whether the response was generated more than five minutes ago, and if so, it won't authenticate the user. This check is performed in the server on top of a normal test that verifies that the response is not expired.

Summary

This attack doesn't rely on a vulnerability in SAML 2.0. It's not a vulnerability in AWS/ADFS, nor in any other service or identity provider.

Golden ticket is not treated as a vulnerability because an attacker has to have domain admin access in order to perform it. That's why it's not being addressed by the appropriate vendors. The fact of the matter is, attackers are still able to gain this type of access (domain admin), and they are still using golden tickets to maintain stealthily persistent for even years in their target's domain.

Golden SAML is rather similar. It's not a vulnerability per se, but it gives attackers the ability to gain unauthorized access to any service in a federation (assuming it uses SAML, of course) with any privileges and to stay persistent in this environment in a stealthy manner.

As for the defenders, we know that if this attack is performed correctly, it will be extremely difficult to detect in your network. Moreover, according to the *'assume breach'* paradigm, attackers will probably target the most valuable assets in the organization (DC, AD FS or any other IdP). That's why we recommend better monitoring and managing access for the AD FS account (for the environment mentioned here), and if possible, auto-rollover the signing private key periodically, making it difficult for the attackers.

In addition, implementing an endpoint security solution, focused around privilege management, like CyberArk's Endpoint Privilege Manager, will be extremely beneficial in blocking attackers from getting their hands on important assets like the token-signing certificate in the first place.

References:

- https://aws.amazon.com/blogs/security/how-to-set-up-federated-api-access-to-aws-by-using-windows-powershell
- https://aws.amazon.com/blogs/security/enabling-federation-to-aws-using-windows-active-directory-adfs-and-saml-2-0/
- https://docs.microsoft.com/en-us/azure/active-directory/develop/active-directory-single-sign-on-protocol-reference