Twitter Thread by Steve Syfuhs

Steve Syfuhs
@SteveSyfuhs



Have you ever had an app that authenticated users, and then thought wouldn't it be great if it could act as that user for the services it has to call later?

This has historically been called Impersonation, Delegation, Act As, or On Behalf Of depending on protocols in play.

Fundamentally the idea is simple. The user projects their identity to some remote application (i.e. authenticate). The app can't do anything with that projection other than accept it. Delegation is the act of granting that app the right to project the identity further downstream.

There's a fundamental requirement with this though. The method of projecting (authing) the user has to rely on a trusted third party. In other words using something like federation or protocols like Kerberos. This is in contrast to just passing around creds. You might wonder why.

It's because if the app in the middle has your password it can just use your password for downstream stuff, ad infinitum. Trusted third party auth protocols by their nature like having 1:1 relationships with all parties. Unfettered projection complicates things.

I keep saying projecting when I technically mean authenticating. You might wonder why. Authentication is the act of verifying a credential, where that credential could be a password, or a token, or a ticket. Most people associate it with just the first step: the user password.

With delegation there's usually multiple forms of credentials in play. However, the identity represented by those credentials remains the same. Hence projecting the identity.

There's another fundamental requirement with delegation. The middle app must be explicitly granted the right to delegate to other things. This is, I hope, somewhat obvious to everyone that it's for enforcing security. Let me explain.

Authentication is about building a trusted relationship (for some definition of trust) between two parties. When I ask Active Directory for a ticket to a web server I'm relying on the web server to trust AD, and by extension trust me. I project my identity to that server.

If the web server could take that projection and act as me for other services, that original trust is broken because AD hasn't blessed the relationship between me and that other service.

This right to delegate can come in many forms and is primarily dictated by the protocol in play. There are two basic types though.

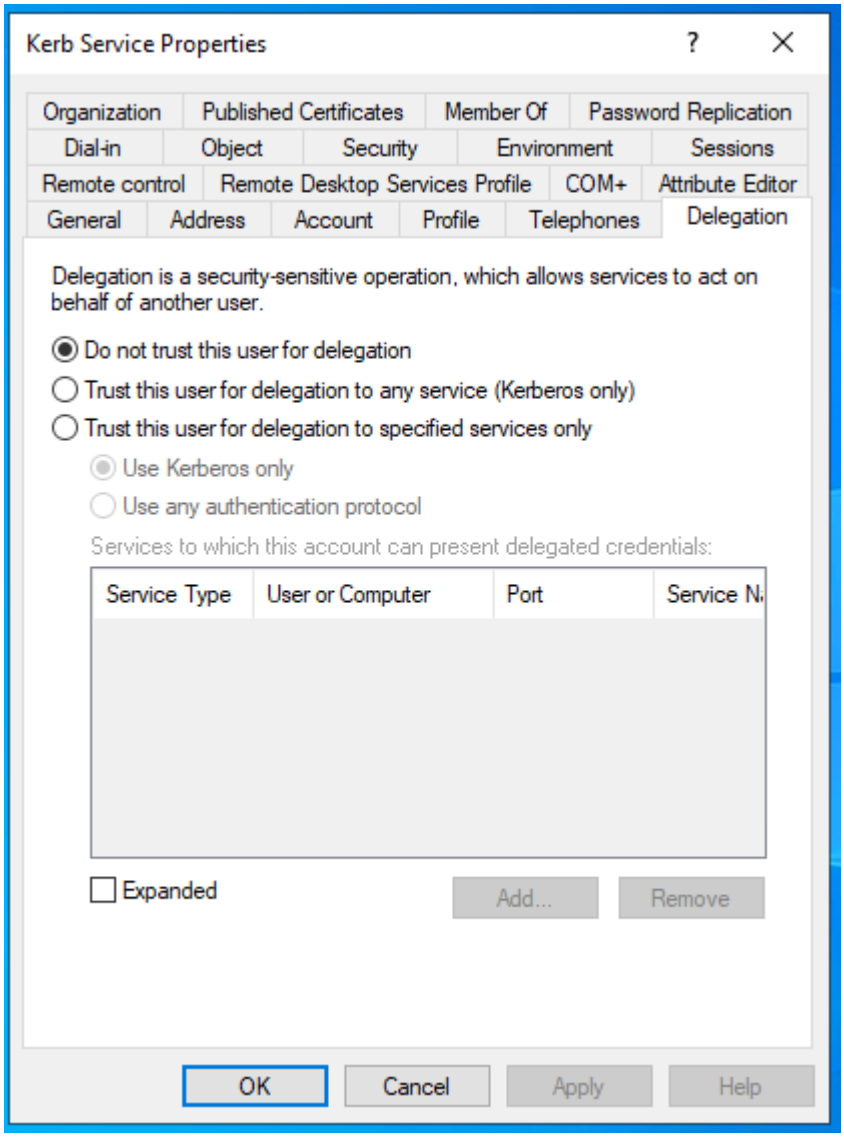
First, there's the right to delegate to anyone. Meaning this middle app can project to any app whatsoever and there's no (or little) limit on it.

Second, there's the right to delegate to specific apps. Meaning the middle app is only allowed to project your identity to a known list of apps. This particular method is preferred for, again I hope, obvious security reasons. Most delegation services stick to this form nowadays.

There are other forms, but they all kinda fit the above patterns.

There are different implementations of delegation. Active Directory calls it delegation. Federation protocols like WS-Fed/Trust call it ActAs. OAuth calls it delegation or impersonation, Azure AD calls it on-behalf-of. Let's look at AD and Azure AD.

Many folks know about delegation because of this screen in Active Directory. Many people have come to hate this screen. This screen applies to that middle web server.



User A projects their identity to web server B, and B projects A's identity to web service C. A => B => C. The screen from before applies to B. That screen lets you control what form of delegation you want to allow.

The first option is obvious. B isn't allowed to delegate to anyone, period.

The second option is mode 1: B can delegate to whoever it wants. Could be C, could be D, could be Y, could be Z.

The third option is mode 2: B can only delegate to C if it's in the list below it.

But the third option has a catch: you can only delegate when using Kerberos. So option 3b was added: you can delegate to whoever is in this list without authenticating. Whaaaaaat the eff? We'll come back to this.

So AD relies on Kerberos for delegation. The first option is called unconstrained delegation. This is the easiest mode to wrap your head around because what you're doing is giving your client TGT to the web server, so the server can do whatever it wants with it.

That's literally all it's doing. The client is informed that the server uses unconstrained delegation so the client copies the user TGT, stuffs it into the AP-REQ authenticator, and fires it off to the target server.

Once the server receives the AP-REQ, it decrypts it and creates a logon session and ticket cache. The TGT is stuffed into the cache. When the server needs to make a call as the user it uses the TGT to request a ticket to whatever is next. https://t.co/w4aqAbhIAq

Incidentally this is why Credential Guard blocks unconstrained delegation on the client. Unconstrained delegation is Stealing-your-TGT-as-a-Service. https://t.co/DM0ikPJEcq

Next is what's called constrained delegation. You provide a list of services web server B is allowed to project the identity to. You might think okay, it's just like unconstrained where they copy the TGT over, and the web server has to ask what it can project to, but it's not.

We don't live in a world where we can blindly trust web server B. If B is even a little untrustworthy and it managed to get a hold of a TGT for the user, it's game over for that user. We in no uncertain terms want that TGT loose on the network. So what do?

Well, we don't do Kerberos. ■

More precisely what we do is use a protocol that is an extension to Kerberos called Service-for-User [MS-SFU] or s4u. https://t.co/D7TAALa6h8

S4u is a way for web server B to safely request a ticket to service C on behalf of the user. It does this by using the service ticket user A handed to server B as an evidence ticket. Huh?

Imagine you're web server B. You have an identity "webserverb$". You have a password, and AD knows who you are. You can receive Kerberos service tickets. AD encrypts them to your password. Nothing particularly fancy about this. It's basic Kerberos. https://t.co/uNgeFphNoU

Now, you receive a ticket from user A and decrypt it. You create the SSO logon session for the user and tell the web app it's user A. Now the web app wants to communicate to service C as that user so it asks for a ticket to service C.

Web server B has a logon session for user A, so web server B checks for a TGT. No TGT because we're constrained. Okay, let's try S4U. To do this, the web server makes a request to the domain controller for a service ticket to C.

The server has an identity on the network, therefore it has a TGT, and it's allowed to ask for service tickets. Again, plain old Kerberos. However, the identity in the service ticket would be server B. So server B adds the service ticket it received from A as an evidence ticket.

Well now this is interesting to the domain controller. It receives a ticket request from B for C and also a service ticket from A FOR B, plus some flags to say do S4U. So the DC checks the delegation list. Is B allowed to delegate to C? Yes, okay give B a ticket to C as A.

I promise you aren't having a stroke. The way to follow this is that B needs to prove it's allowed to get the ticket to C on behalf of A, otherwise B could be evil and impersonate whoever they want.

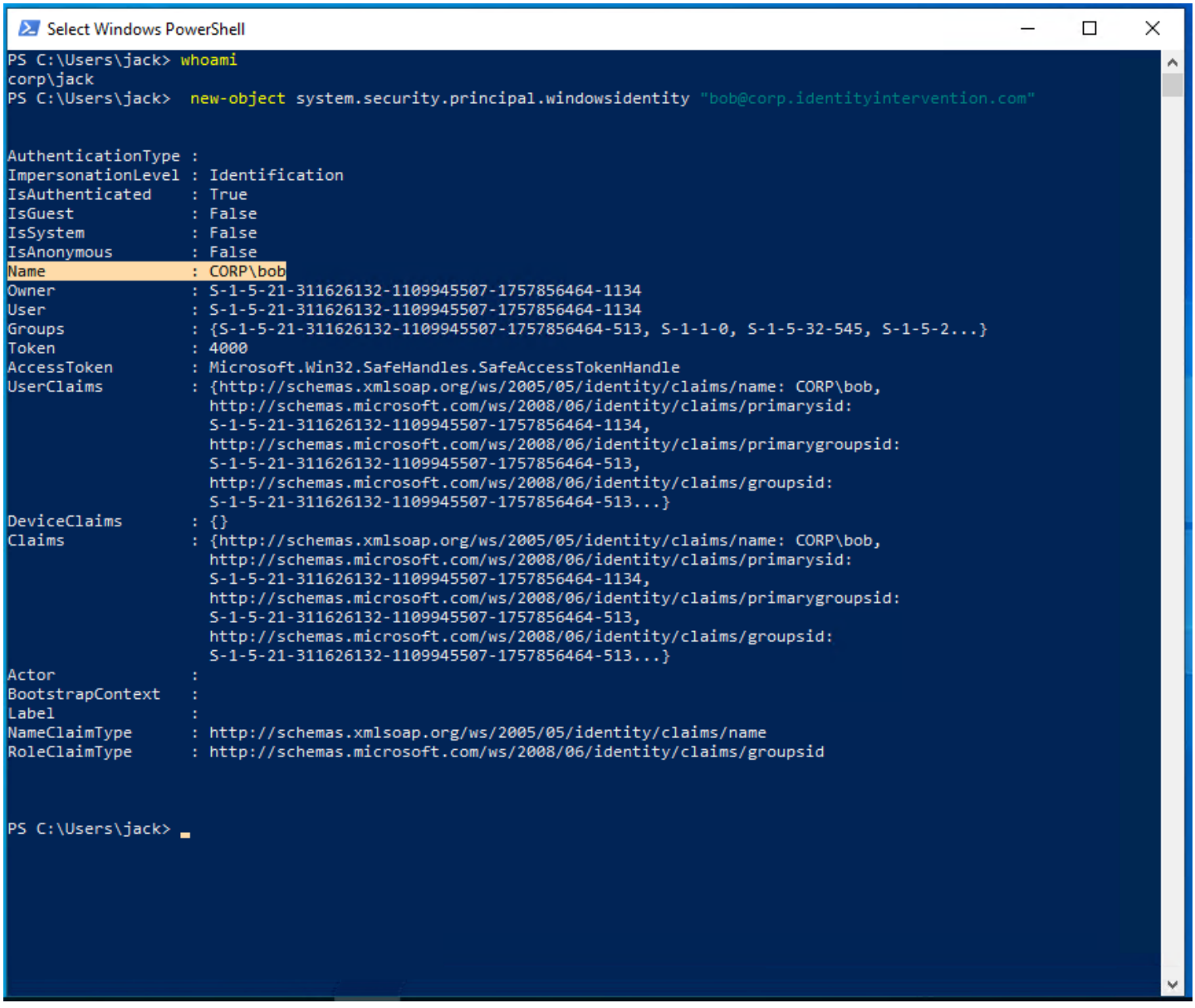
Once web server B receives the ticket, it bubbles it up to the web app, and the web app stuffs the ticket into the request for web service C. C receives it, decrypts it, and sees it's for user A. And we're back to how SSO works. https://t.co/w4aqAbhIAq

But remember how I said there's that other mode, where it doesn't require authentication? Yeah about that. We call it Protocol Transition. This exists so web server B can authenticate users using a protocol Active Directory doesn't understand, while still impersonating them.

Consider if web server B supports SAML. AD doesn't understand SAML, but the web app wants to impersonate the identity projected through SAML so when they call web service C they're acting as user A. This can work through protocol transition.

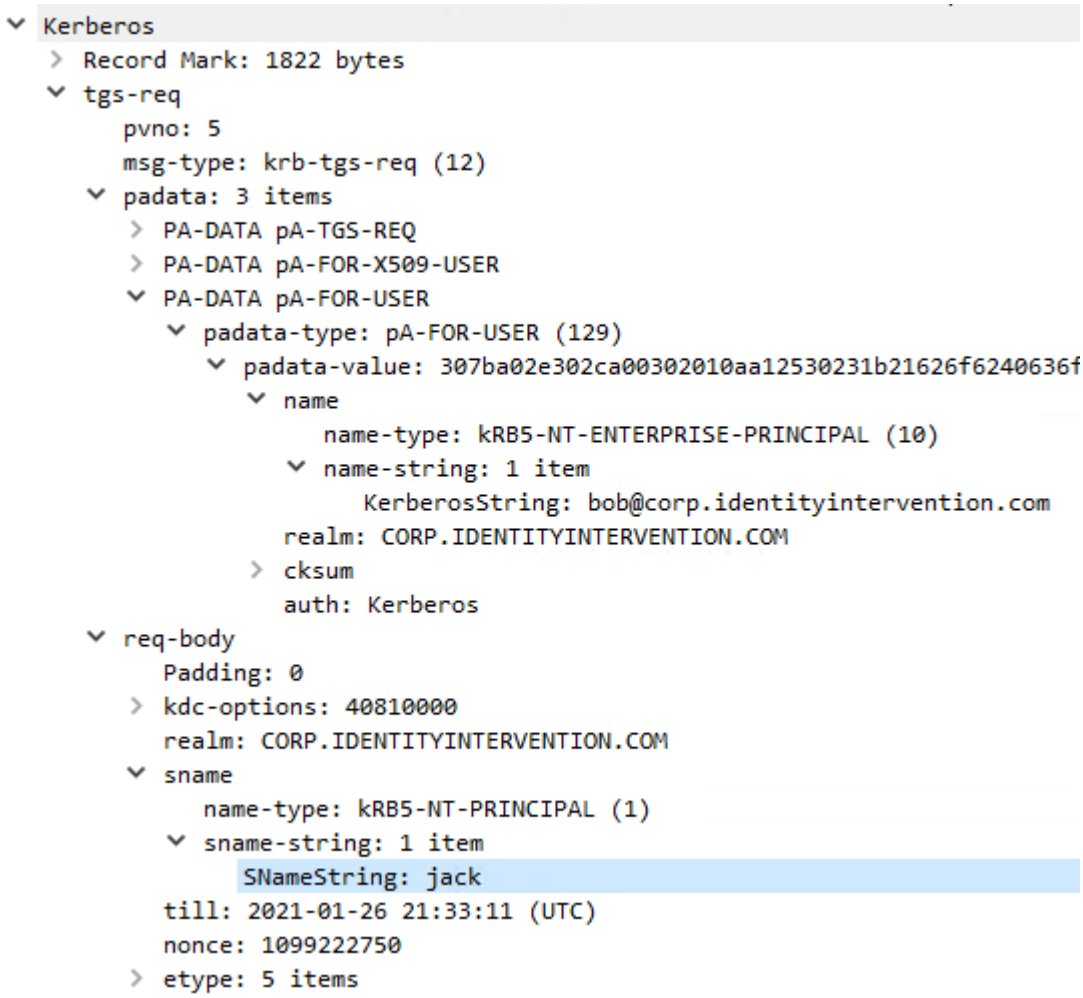
It uses the same protocol extension -- S4U. The difference is that the evidence ticket isn't real. Or rather, it's an evidence ticket issued to the server itself. Huh?

There's a sub-protocol in S4U called service-for-user-to-self or s4u-self. This is an incredibly useful protocol, because an application can request a service ticket to *itself* on behalf of any user. It's useless for auth, but still contains the user identity. Give it a try.



new-object https://t.co/kuyNLuCzxr.principal.windowsidentity "bob@corp.identityintervention.com"

It's a TGS-REQ to me, with some bits that says please get on behalf of bob.



So for just a name you get a ticket to yourself. A ticket to yourself can get your constrained delegation. Put the two together and you have s4u-proxy, AKA protocol transition!

Of course, here's the problem: protocol transition is increeeedibly dangerous. You've just granted web server B the rights to impersonate whoever they want. That's worse than unconstrained delegation. Oops. As such you've effectively granted web server B the right to act as a DC.

This is, incidentally why selecting that option tends to make things "just work".

Please, please, pleeeeeease be incredibly careful if you choose to use it. Any server configured for protocol transition should have the same security as a DC. No joke.

Anyway, there's one more thing with AD delegation that isn't on that screen. It's something called Resource-Based Constrained Delegation. Huh?

It's basically regular constrained delegation, but the approval list is flipped. The idea is that instead of saying web server B is allowed to delegate to web service C, it says web service C is allowed to be delegated from web server B.

This actually makes more sense from a security perspective. The thing you're protecting is web service C, so why are the controls on B? The armed money delivery transports don't dictate which vaults they can go in to, the banks dictate which transports are allowed into the vault.

The other useful thing about RBCD is more of a practical one. Regular constrained delegation doesn't work across forests because of how the protocol happens to work. There isn't enough authz details in it to identify the service because SPNs are only unique within forests.

RBCD relies on SIDs. Web service C has a list of SIDs that can delegate to it, and when web server B makes a request to the other forest, it includes web server B's SID.

But enough about that. Let's look at another form of delegation: Azure AD on-behalf-of.

Azure AD-based apps are an evolution of AD-based apps. They still have similar, or higher, security requirements. As web server B I want to project user A's identity to service C.

The difference is in the choice of protocols. Azure AD relies on OAuth2 for authentication and authorization. A client authenticates user A and gets a JWT access token signed by AAD to web server B. Web server B validates the ticket and the app now knows it's user A, so says AAD.

But the web app wants to access another resource, say Microsoft Graph. The web app can use it's own identity, or it can use the identity of the user. You could keep it simple and use the app identity, but then you need to grant that identity access to all users resources.

That's a fairly broad set of permissions, and attacking that app gets you everything in Graph. That's bad.

But if you use the user identity, the app doesn't need anything special, and the app can operate as just that single user. That's good.



Protocol-wise this is an extension to OAuth called on-behalf-of. https://t.co/oc2eBhBcao

Not surprisingly it kinda sorta acts a lot like s4u. User A gets a token to server B from AAD. Server B has it's own identity and authenticates to AAD, providing user A's access token to B as an evidence token (assertion).

AAD checks the list of apps that B is allowed to delegate to, and if C is in the list, it issues a token to B on behalf of A for C.

This is actually pretty cool. First, it's way easier to follow along because it's all HTTP instead of binary encoded ASN.1 goop. Second, it supports multiple token types. Ergo it supports multiple protocols.

That means you can transition between OAuth and (say) SAML without turning your app into a major target, because every exchange and token is still authenticated. Awesome.

Third, you can enforce consent. This shifts authorization a bit so the user is now in control over who can project and forward their identity. That's a nice privacy win.

So obviously there's been a delegation evolution. From AD unconstrained, all the way to OAuth on-behalf-of. Hopefully it's gotten easier and more secure along the way. /fin

Archived https://t.co/whL0Lc8ZOi