Kerberos простыми словами / Хабр

habr.com/ru/articles/803163

Юрий Строжевский

ystr 27 мар 2024 в 10:29

Kerberos простыми словами

Информационная безопасность*Сетевые технологии*

Введение

Несмотря на то, что уже существует множество различных статей про Kerberos, я всё-таки решил написать ещё одну. Прежде всего эта статья написана для меня лично: я захотел обобщить знания, полученные в ходе изучения других статей, документации, а также в ходе практического использования Kerberos. Однако я также надеюсь, что статья будет полезна всем её читателям и кто-то найдёт в ней новую и полезную информацию.

Данную статью я написал после того, как сделал собственную библиотеку, генерирующую сообщения протокола Kerberos, а также после того, как я протестировал «стандартный клиент» Kerberos в Windows — набор функций SSPI. Я научился тестировать произвольные конфигурации сервисов при всех возможных видах делегирования. Я нашёл способ, как выполнять пре-аутентификацию в Windows как с применением имени пользователя и пароля, так и с помощью PKINIT. Я также сделал библиотеку, которая умещает «стандартный» код для запроса к SSPI в 3–5 строк вместо, скажем, 50-ти.

Хотя в названии статьи фигурирует «простыми словами» однако эта статья предполагает, что читатель уже имеет какое-то представление о Kerberos.

Что есть в данной статье

- Простое объяснение основ работы Kerberos;
- Простое объяснение аутентификации между доменами;
- Описание реализации передачи сообщений Kerberos как по UDP, так и по TCP;
- Объяснение что такое «сервис» в понятии Kerberos и почему у «сервиса» нет пароля;
- Объяснение почему в абсолютном большинстве случаев SPN служит просто для идентификации пользователя, из-под которого запущен «сервис»;
- Как система понимает, к какому типу относить то или иное имя принципала;
- Рабочий код на C++, реализующий алгоритм шифрования Kerberos с AES (код реализован с применением CNG);
- Как передать X.509 сертификат в качестве credentials в функцию AcquireCredentialsHandle (PKINIT);
- Как сделать так, чтобы «сервис» понимал чужие SPNs (одновременная работа с credentials от нескольких пользователей);
- Почему GSS-API является основой для всего процесса аутентификации в Windows (да и в других ОС тоже);
- Описание (со скриншотами из Wireshark) того, как именно передаётся AP-REQ внутри различных протоколов (LDAP, HTTP, SMB, RPC);
- Простое описание для каждого из типов делегирования Kerberos;
- Код для тестирования каждого из типов делегирования, который может быть выполнен на единственной рабочей станции, без какой-либо конфигурации кучи вспомогательных сервисов;
- Описание User-To-User (U2U), а также код с возможностью протестировать U2U;
- Ссылки на различный мой код, как то: реализация «whoami на S4U», реализация klist, библиотеки XKERB для запросов к SSPI и прочее;

Для чего нужен Kerberos

Для начала представим, что на компьютере (любом) в сети запущена программа. Эта программа может общаться с другими программами в сети с помощью какого-то произвольного протокола. Представим также, что эта программа может предоставлять какой-то «сервис» другим программам: например, выполнить сложение «2+2» и затем переслать результат обратно. Пока «сервис», предоставляемый этой командой, ограничивается только контекстом самой программы, то всё просто. Однако теперь представим, что в качестве «сервиса» эта программа позволяет считывать какой-то локальный файл и затем передать содержимое этого файла по сети. В этом случае эта программа уже столкнётся с понятием доступа к необходимому файлу.

Как известно, в Windows доступ к файлу определяется на основе security descriptor, где прописаны какие именно пользователи и группы имеют возможность выполнять определённые операции над данным файлом. Информация же о текущем конкретном пользователе и его группах содержится в так называемом access token (далее просто «токен»), который присваивается каждому отдельному процессу и потоку. Обычно такой токен получается пользователем при первичном логине на рабочую станцию. И все процессы, которые будут запущены от имени залогиненного пользователя, будут «наследовать» этот токен.

Теперь вернёмся обратно к той самой программе, которая запущена на локальном компьютере и предоставляет «сервис» по чтению контекста локального файла. Предположим, что ей поступил запрос на чтение локального файла А. Сама программа выполняется из-под пользователя, который находится, например, в группе «Users» на данном компьютере. Однако доступ к файлу А возможен только для пользователей из группы «Administrators». То есть в том случае, если эта программа будет использовать свой «базовый» токен, то для неё доступ к файлу А будет закрыт. Но ведь эта программа предоставляет «сервис» каким-то другим программам, которые, в свою очередь, могут быть запущены уже под пользователями, входящими в группу «Administrators». То есть вроде как логично, что если «сервис» запрашивается от имени пользователя, которому разрешён доступ к удалённому файлу, то он ему должен быть предоставлен. Что же нужно для того, чтобы это стало возможным?

На самом деле всё, что нужно — это сделать новый access token на имя того пользователя, от которого пришёл внешний запрос на этот «сервис». Далее программа, предоставляющая данный «сервис», должна просто сделать новый поток и присвоить этому новому потоку вновь сделанный access token. Теперь для локального компьютера всё, что будет сделано в рамках работы данного потока будет сделано от имени удалённого пользователя, который первично сделал запрос на «сервис». Но так или иначе для создания нового access token на «сервис» в качестве входных данных должна прийти вся информация, необходимая для этого.

Теперь рассмотрим ещё более сложную ситуацию. Допустим, на «сервис», который запущен на компьютере А пришёл запрос «запроси сервис на компьютере Б». В этом случае, опять же, есть две возможности: либо запросить сервис Б от имени пользователя, под которым запущен сервис А, или запросить сервис Б от имени пользователя, который сформировал первичный удалённый запрос к сервису А. Если мы хотим, чтобы был реализован второй вариант, то получается, что мы должны «делегировать» сервису А возможность представляться пользователем, который сформировал к нему запрос. Конечно, и в ранее рассмотренном случае с доступом к локальному файлу также был факт «делегирования», но там был случай «локального делегирования», только в рамках локального компьютера. Здесь же сервису А должно быть делегировано полномочие представлять какого-то пользователя в рамках взаимодействия с другими сервисами по сети. Так что для возможности «делегирования» в протоколе «сервиса» также должны быть какие-то данные.

Kerberos является протоколом аутентификации. Есть три различных понятия: идентификация, аутентификация и авторизация. Идентификация — это когда ты подходишь к какому-то пропускному пункту и говоришь охраннику «я Вася Пупкин». Аутентификация — это когда кто-то «авторитетный» заверяет вашу идентификацию, например рядом стоящий другой охранник говорит первому «да, я знаю этого парня, это точно Вася Пупкин». Авторизация — это когда этот самый охранник посмотрит твои документы, потом сверится со своим личным списком лиц, которым разрешён проход, и скажет «Васи Пупкина нет в списке, вы не проходите». То есть в данном примере идентификация прошла успешно, аутентификация также была выполнена, а при авторизации произошёл отказ.

В Kerberos для аутентификации (подтверждения идентификационной информации) используется передача зашифрованных сообщений. Если конечный получатель сообщения смог корректно расшифровать это сообщение, то однозначно считается, что информация, которую содержит сообщение, является верной. Существуют механизмы дополнительной проверки корректности этой информации в виде нескольких типов «подписей», но для текущего повествования это маловажно.
Для шифрования идентификационных сообщений используются ключи шифрования, которые формируются специальным образом на основе текстовых паролей конечных получателей. Предполагается, что эти пароли известны только конечным получателям идентификационных сообщений, а также некоторому общему для домена центру хранения паролей. Таким образом, для того, чтобы пользователь А был идентифицирован пользователем Б пользователь А просит центр хранения паролей сформировать идентификационное сообщений, которое было бы зашифровано на шифровальном ключе пользователя Б. Центр хранения паролей формирует такое сообщение, и пересылает его обратно пользователю А. Пользователь А не может расшифровать или изменить это сообщение (оно зашифровано на ключе пользователя Б). Далее пользователь А может переслать это идентификационное сообщение пользователю Б, он его сможет успешно расшифровать и прочитать хранящуюся в данном сообщении идентификационную информацию.

В качестве «идентификационной информации» вполне достаточно передавать только уникальное имя пользователя. Однако в Windows пользователь идентифицируется не только своим именем, а также набором групп пользователей и набором привилегий. Можно было бы сделать так, чтобы сервис после расшифровки идентификационного сообщения сам посылал куда-то запросы, на которые бы ему возвращались все дополнительные данные пользователя. Однако был выбран путь включения в идентификационное сообщение полной информации о «предмете идентификации»: полный список групп, а также другой вспомогательной информации. Привилегии «выдаются» каждому пользователю или группе на каждом компьютере отдельно, так что добавление привилегий в access token происходит уже в процессе формирования access token на машине «сервиса». Кстати, в Active Directory «предмет идентификации» называется «принципал» (principal).

В Active Directory пользователями идентификационных сообщений могут быть любые сущности, у которых возможно существование пароля. К таким сущностям относятся как пользователи домена, так и компьютеры, которые в данный домен входят. Использование компьютеров как пользователей идентификационных сообщений удобно так как нужно, чтобы «сервисы» работали даже тогда, когда ни один пользователь не работает за компьютером.

Также в Active Directory возможна аутентификация между пользователями из «дружественных» доменов. Технически это реализуется путём добавления в каждый из доменов специальных пользователей, имена которых совпадают с именем дружественного домена. При необходимости пользователя из домена А получить доступ к сервисам из домена Б пользователь сначала посылает запрос к своему контроллеру домена. Там формируется аутентификационная информация, однако она шифруется не на ключе начального контроллера домена, а на ключе того пользователя, имя которого совпадает с именем «дружественного» домена. Далее пользователь просто пересылает полученную аутентификационную информацию уже контроллеру домена Б и процесс аутентификации продолжается стандартным образом.

Техническая реализация Kerberos

Теперь опишем реальную структуру, которая обслуживает протокол идентификации Kerberos в Windows Active Directory. В Active Directory для каждого домена существует Key Distribution Center (KDC). Именно эта сущность отвечает за хранение паролей всех пользователей и компьютеров в данном домене. В документе RFC 4120 применяется также разделение на Authentication Service (AS) и Ticket-Granting Service (TGS), однако в Active Directory все эти сущности объединены. Для простоты будем считать, что все коммуникации осуществляются с KDC.

Все сообщения в Kerberos кодируются с помощью ASN.1. Описание схем ASN.1 для различных видов сообщений можно найти в RFC 4120 и других документах. Коммуникации с KDC в Active Directory осуществляются либо с использованием протокола TCP (KDC слушает на порту 88), либо с использованием протокола UDP (и опять KDC слушает на порту 88). Найти на каком именно адресе работает сервис Kerberos можно с помощью команды «nslookup -type=SRV _kerberos._tcp.<имя_домена>» или для UPD с помощью команды «nslookup -type=SRV _kerberos._udp.<имя_домена>». Если сообщения передаются по протоколу TCP, то перед каждым сообщением передаётся 4 байта, содержащие общую длину сообщения. Если передача происходит по UDP, то передаётся просто чистое сообщение, без информации о длине.

Все коммуникации с KDC начинаются с посылки сообщения KDC-AS-REQ. В данном сообщении присутствует информация о различных флагах (об этом позже), имени принципала, имени домена, имени принципала, который является специальным и отвечает за выдачу идентификационной информации (оно всегда имеет значение «krbtgt»), а также о списке поддерживаемых схем шифрования. Насчёт «схем шифрования» необходимо дать дополнительное пояснение. Дело в том, что в KDC-AS-REQ передаётся набор тех схем шифрования, которые может поддержать (зашифровать/расшифровать) данный конкретный клиент. На KDC, в свою очередь, может быть какой-то другой набор поддерживаемых схем шифрования, и, потенциально, они могут не совпасть со схемами на клиенте. В этом случае коммуникации между таким клиентом и KDC будут невозможны.

В ответ на получение сообщения KDC-AS-REQ KDC может возвратить как зашифрованную идентификационную информацию для упомянутого в запросе принципала (сообщение KDC-AS-REP), так и ошибку (KRB-ERROR). В KRB-ERROR будет передаваться, прежде всего, код ошибки, а также возможна передача дополнительной информации. Обычно в современных системах код ошибки будет 25: требуется пре-идентификация. Под пре-идентификацией подразумевается процесс, в ходе которого запрашивающая сторона проверяется на то, что она действительно знает ключ шифрования того принципала, для которого запрашивается идентификационная информация. Также в случае кода ошибки 25 в KRB-ERROR передаётся дополнительная информация, как то: 1) специальное текстовое сообщение, которое используется при генерации ключа шифрования (salt); 2) идентификатор предпочтительной схемы шифрования; 3) перечень способов пре-идентификации, которые поддерживаются данным KDC. Необходимо заметить, что обычно KDC поддерживает несколько способов пре-идентификации, таких как пре-идентификация по паролю, пре-идентификация по сертификату (PKINIT), пре-идентификация с использованием расширения FAST (RFC 6113) и так далее. Однако базовым способом пре-идентификации является идентификация по паролю. Для того, чтобы клиент прошёл эту пре-идентификацию, он должен зашифровать специально сформированные данные, содержащие текущую временную метку. То есть клиент просто должен каким-то образом получить текущее время, затем сформировать специальное сообщение, и затем зашифровать его с использованием пароля того принципала, для которого он запрашивает у KDC идентификационную информацию. В случае с использованием пре-идентификации по сертификату (PKINIT) процесс практически такой же за исключением того, что шифрование производится с использованием закрытого ключа сертификата принципала.

После формирования пре-идентификационного сообщения формируется новый вариант сообщения KDC-AS-REQ, который содержит всю информацию из первого варианта сообщения плюс информацию о зашифрованном текущем времени на клиенте. Эта информация располагается внутри массива PA-DATA (pre-authentication data).

В ответ на KDC-AS-REQ в котором содержится пре-идентификационная информация KDC, присылает KDC-AS-REP, который содержит идентификационную информацию по запрошенному принципалу. Эта идентификационная информация зашифрована на ключе KDC и служит для реализации функционала Single Sign-On (SSO). В большинстве случаев эту идентификационную информацию называют «Ticket-Granting Ticket» (TGT). То есть для всех дальнейших запросов к KDC пользователю не нужно будет всегда использовать пароль: он будет просто посылать свою идентификационную информацию KDC, затем KDC сможет её расшифровать и использовать эту информацию во вторичных запросах. Нужно заметить, что в KDC-AS-REP присылается как идентификационная информация, которую клиент не может расшифровать, так и дополнительная информация, зашифрованная на ключе клиента и к которой он имеет доступ. В этой дополнительной информации, в частности, находится случайно сгенерированный сессионный ключ, который клиенту теперь будет необходимо использовать в дальнейших коммуникациях с KDC. Использование сессионного ключа вместо заранее известного ключа, генерируемого из пароля пользователя, потенциально повышает безопасность коммуникаций. Кстати, на самом деле KDC нигде не хранит этот сессионный ключ. Вместо этого он просто добавляет его в ту идентификационную информацию, которую присылает в KDC-AS-REP. Когда клиент выполнит следующий запрос к KDC он пошлёт эту же идентификационную информацию, которую KDC сможет расшифровать и достать оттуда нужный сессионный ключ.

Флаги в KDC options

Основные флаги, поддерживаемые в Windows, описаны в документе по следующей ссылке. Здесь я расскажу о наиболее используемых флагах.

Начнём с флага CANONICALIZE. Если данный флаг установлен в запросе от клиента, то это означает, что клиент ожидает (и может обработать) изменение имени клиента и сервиса, а также изменение типа этого имени в ответном сообщении. То есть при посылке TGS-REQ с именем клиента «client@server» в TGS-REP может быть уже «server\client».

Флаг RENEWABLE устанавливается на KDC при формировании первичных TGT. Если это флаг установлен, то клиент может обновлять свой TGT, без полного прохождения процедуры пре-аутентификации. Делается это путём повторной посылки сообщения TGS-REQ к сервису «krbtgt», установленным флагом RENEWABLE в запросе и первичным TGT в PA-DATA. Далее KDC просто декодирует старый TGT, скопирует его содержимое в новый TGT, обновить время действия нового TGT, а также сформирует новый сессионный ключ.

Флаг OK-AS-DELEGATE находится в поле «flags» зашифрованной части TGS-REP (не в kdc_options). Флаг OK-AS-DELEGATE формируется на стороне KDC, и приходит в TGS-REP когда выполняется запрос к сервису, для которого включён режим «неограниченного делегирования» (более подробно про данный режим далее в статье). При получении TGS-REP с установленным OK-AS-DELEGATE стандартный Kerberos клиент Windows автоматически формирует TGS-REQ к сервису «krbtgt» с установленными флагами FORWARDABLE и FORWARDED. В этом новом TGS-REQ также прикладывается первичный TGT клиента. В ответ KDC формирует TGS-REP, в котором содержится TGT с установленным флагом FORWARDED и именно этот новый TGT используется далее для формирования AP-REQ к необходимому сервису. Таким образом, если у вас есть TGT с установленным флагом FORWARDABLE и есть сессионный ключ, соответствующий данному TGT, то можно сформировать запрос, который возвратит вам новый TGT с установленным FORWARDABLE. Флаг FORWARDABLE необходим при реализации делегирования на сервисах, без этого флага KDC будет отвергать запросы от сервисов на получение билетов на имя клиента.

Далее для продолжения рассказа о технических аспектах реализации Kerberos необходимо прерваться и рассказать о том, как реализованы «сервисы» в Active Directory.

Сервисы

Под «сервисом» понимается какая-то программа, предоставляющая какой-то функционал удалённым или локальным (по отношению к компьютеру, на котором этот «сервис» запущен) пользователям. Нужно заметить, что в Windows есть даже отдельная оснастка для MMC, которая показывает «Сервисы». Но на самом деле те «сервисы», которые отображаются в этой оснастке, не относятся к «сервисам» в понятии Kerberos. Для Kerberos сервисом будет считаться любая программа, даже запущенная из-под обычного пользователя. То есть «сервис» из оснастки MMC может быть также «сервисом» для Kerberos, однако это не является обязательным условием.

Как я описал ранее, для аутентификации в Kerberos используются зашифрованные сообщения. Для шифрования в адрес какого-то принципала необходимо, чтобы у этого принципала был какой-то пароль. Пароли в домене Active Directory могут быть только у пользователей или компьютеров. «Сервисы» сами по себе паролей не имеют. Вместо этого «сервис» использует идентификационную информацию (имя пользователя и пароль) того пользователя, из-под которого этот «сервис» в настоящий момент запущен. И, как следствие, все «сервисы», которые запущены из-под одного и того же пользователя, будут использовать один и тот же пароль. Если «сервис А» запущен из-под пользователя А на компьютере 1, то если запустить «сервис Б» на компьютере 2 и запустить его опять из-под пользователя А, то оба сервиса будут использовать один и тот же пароль. Для более «продвинутых» читателей приведу ещё пример: все сервисы, запущенные на любом компьютере из-под одного и того же аккаунта gMSA используют один и тот же пароль.

Для Kerberos «сервис» тоже является принципалом. То есть «сервис» может использовать идентификационную информацию других принципалов, а также передавать свою идентификационную информацию. Однако так как «сервис» использует идентификационную информацию того пользователя, из-под которого он запущен, то фактически «сервис» можно назвать «фантомным» принципалом. В первичных стандартах Kerberos «сервис» имеет собственное имя (Service Principal Name, SPN), однако в Active Directory это имя, фактически, используется только для поиска пользователя, который в настоящий момент владеет данным SPN и вместо имени «сервиса» может быть использовано имя того принципала, из-под которого данный сервис запущен.

В Active Directory имя «сервиса» (SPN) также используется, но только для того, чтобы найти имя того принципала, из-под которого данный «сервис» запущен. Для того, чтобы это было возможно SPN должно быть уникальным в рамках всего домена. Так как обычно «сервисы» запускаются из-под учётных записей компьютеров, то имя SPN для любого такого «сервиса» должно (по правилам Microsoft) содержать имя компьютера, на котором этот «сервис» запускается. Но если вы для компьютера COMP1 добавите новый «сервис» и зададите ему SPN вида «service1/COMP2», то Active Directory это тоже пропустит. Главное, чтобы такого имени ранее в домене не существовало. Возможно, кто-то подумает, что если SPN=service1/COMP2, то теперь KDC при запросах также будет использовать и пароль для COMP2 — нет, KDC будет использовать пароль того пользователя, у которого в атрибуте servicePrincipalName находится искомое SPN. Кстати, не смотря на официальную документацию, в которой описывается как именно должны именоваться SPN в Active Directory (ссылка) у меня получалось добавлять (и в дальнейшем использовать) SPN вида «S1/S1». Повторюсь — главное, чтобы имя «сервиса» было уникальным в масштабе текущего домена.

Любой «сервис» может быть запущен как из-под системной учётной записи (записи компьютера), так и из-под учётной записи произвольного пользователя. Но если меняется пользователь, из-под которого запускается какой-то «сервис», то для корректной работы Kerberos необходимо просто удалить SPN этого сервиса из атрибута servicePrincipalName у одного пользователя/

компьютера и добавить этот SPN в аналогичный атрибут другого пользователя/компьютера. Отличием между запуском из-под системной учётной записью и записью обычного пользователя будет только набор привилегий, которыми обладает пользователь: для полноценной работы «сервисам» зачастую необходимо, чтобы пользователь, в контексте которого выполняются «сервисы», имел такие привилегии как SeTcbPrivilege и SeImpersonatePrivilege. Однако если выдать права на эти привилегии какому-то другому пользователю на данном компьютере, то «сервис», выполняемый от его имени, будет иметь точно такие же возможности, как если бы он выполнялся от имени системной учётной записи.

Также необходимо заметить, что некоторые имена «сервисов» являютс предопределёнными, и другие «сервисы» ожидают, что «сервис» с таким именем будет предоставлять определённые функции. Так «сервис» с именем «krbtgt/<имя домена>» будет предоставлять сервисы KDC, «сервис» с именем «host/<имя компьютера>» предоставляет возможность логина на компьютер, «сервис» с именем «cifs/<имя компьютера>» предоставляет услуги доступа к файловой системе на определённом компьютере по протоколу SMB и так далее. Список таких «предопределённых имён» можно посмотреть тут. Посмотреть, какие «сервисы» зарегистрированы для определённого компьютера можно с помощью команды «setspn.exe -L <ServerName>». Посмотреть, какие «сервисы» зарегистрированы для определённого пользователя можно с помощью команды «`setspn.exe -L <domain\user>`».

Общие правила составления имён принципалов

Перед правилами именования принципалов необходимо пояснить какие же типы имён принципалов бывают в Active Directory. В файле «NTSecAPI.h» объявлены следующие типы имён принципалов:

- KRB_NT_UNKNOWN
- KRB_NT_PRINCIPAL
- KRB_NT_PRINCIPAL_AND_ID
- KRB_NT_SRV_INST
- KRB_NT_SRV_INST_AND_ID
- KRB_NT_SRV_HST
- KRB_NT_SRV_XHST
- KRB_NT_UID
- KRB_NT_ENTERPRISE_PRINCIPAL
- KRB_NT_ENT_PRINCIPAL_AND_ID
- KRB_NT_MS_PRINCIPAL
- KRB_NT_MS_PRINCIPAL_AND_ID

Если в имени присутствует «@», то всё, что находится после «@» считается именем домена (realm name). Имя перед «@» делится по группам на основе присутствия символа «/». Если этих групп 1 (то есть нет символа «/»), то общему имени присваивается тип KRB_NT_PRINCIPAL. Если таких групп 2, то общему имени присваивается тип KRB_NT_SRV_INSTANCE. Если таких групп 3 и более, то вроде как общему имени должен присваиваться тип KRB_NT_SRV_XHST, однако на деле присваивается также KRB_NT_SRV_INSTANCE. Если в общем имени присутствует символ «\», то имени присваивается тип KRB_NT_MS_PRINCIPAL_NAME. Если в имени нет ни одного из символов «@», «\» или «/», то этому имени назначается тип KRB_NT_PRINCIPAL и имя домена берётся из текущих значений. Таким образом, если хотите запросить билет для пользователя, то формируйте имя в виде «name@domain» или «domain\name». Если хотите запросить билет для сервиса, то формируйте имя в виде «part1/[part N]@domain» или просто «part 1/[part N]». Всё решает прямой или обратный backslash.

Все типы имён, кончающихся на «_ID» представляют собой обычный тип (без «_ID»), но с добавленным SID в качестве последнего элемента массива имён в виде строки. Данный SID должен быть в стандартной форме, вроде «S-1–5–500».

Описание алгоритма шифрования Kerberos

На самом деле, как я и писал ранее, Kerberos поддерживает множество различных схем шифрования. Также Kerberos может использовать CMS (Cryptographic Message Syntax, RFC5652), как минимум на этапе пре-идентификации. Однако в настоящее время наиболее используемыми схемами шифрования являются схемы на основе использования стандарта шифрования AES. Само по себе шифрование применяется с использованием стандартного алгоритма AES. Однако формирование ключа шифрования в Kerberos выполняется по достаточно сложной схеме. Реализацию на языке C++ с использованием CNG (Cryptography API: Next Generation) можно найти по этой ссылке. Для общего понимания необходимо знать, что для дополнительного усложнения Kerberos использует так называемые key usage numbers: 4-байтовые значения, которые специфичны для каждой фазы передачи сообщений. Значения для этих key usage numbers приведены в RFC4120, item 7.5.1. Таким образом, для фазы посылки первичного запроса AS-REQ с encrypted timestamp значение для key usage number будет равно 1 (или если писать в виде 4-х байтов, то 0×00 000 001), для фазы передачи AS-REP значение key usage number будет уже 2, и так далее. Знание key usage number критически важно для взлома ключей Kerberos — если указать неверный номер, то ключ никогда не будет взломан, или будет получено ошибочное выходное значение.

Обратно к процессу получения идентификационной информации

Итак, раннее я остановился на том, что клиент Kerberos получил от KDC зашифрованную идентификационную информацию. Но всё, что возможно сделать имея эту идентификационную информацию, это идентифицироваться на KDC. Что же делать, чтобы наш клиент смог идентифицироваться на других принципалах, доступных в домене? Для этого KDC предоставляет сервис, формирующий идентификационную информацию, зашифрованную на ключах необходимых принципалов. То есть процесс аутентификации для данного клиента на произвольном принципале в домене состоит в следующем. Изначально клиент идентифицируется на KDC. Полученная на данном этапе идентификационная информация зашифрована на ключе KDC. Далее «прикладывая» к сообщению эту аутентификационную информацию клиент формирует дополнительный запрос к KDC, в котором указывает имя того принципала, к которому ему необходимо получить идентификационную информацию. Всё, что далее делает KDC это расшифровывает аутентификационную информацию, которую «приложил» клиент, достаёт оттуда идентификационные данные клиент и затем зашифровывает эту же идентификационную информацию, но уже на ключе того принципала, идентификацию для которого запросил клиент. То есть процесс крайне прост: расшифровал данные на одном ключе, скопировал их и зашифровал на другом ключе. Далее вновь зашифрованная идентификационная информация пересылается обратно клиенту, и теперь клиент может переслать её (произвести процесс аутентификации) другому принципалу. Это принципал, в свою очередь, сможет эту аутентификационную информацию расшифровать. Теперь для этого принципала наш клиент считается аутентифицированным.

Нужно также понимать одну особенность KDC: он выдаёт аутентификационную информацию для любого принципала в системе, вне зависимости от того имеет ли конкретный клиент Kerberos туда доступ или нет. Это происходит потому, что Kerberos является протоколом аутентификации, а вопрос доступа уже относится к понятию авторизации.

Технически запрос к KDC на предоставление аутентификационной информации для другого принципала формируется с помощью сообщения KDC-TGS-REQ. Идентификационная информация, которую ранее получил клиент в сообщении KDC-AS-REP, сохраняется в KDC-TGS-REQ в поле «additional-tickets». Также для Active Directory возможно передать эту идентификационную информацию в PA-DATA используя тип PA-TGS-REQ. Имя принципала, для которого клиент хочет получить идентификационную информацию, сохраняется в поле «sname» (service name). Также в KDC-TGS-REQ пересылается ещё одна внутренняя структура, которая имеет тип Authenticator. Эта структура имеет в своём составе время формирования KDC-TGS-REQ, а также продублированное имя принципала из другой части KDC-TGS-REQ. Также Authenticator шифруется на сессионном ключе, который клиент получает вместе с KDC-AS-REP. Использование Authenticator позволяет в какой-то мере противостоять атакам, связанным с повторной передачей данных. Для этого на KDC ведётся учёт всех ранее полученных данных из Authenticator, и если приходит сообщение, в котором все значения совпадают с ранее полученными, то KDC возвращает в ответ ошибку.

В ответ на KDC-TGS-REQ сервис KDC присылает сообщение KDC-TGS-REP, в котором содержится идентификационная информация для нужного принципала. Далее будем обозначать эту идентификационную информацию как «Service Ticket» (ST).

Теперь необходимо пояснить, как же именно происходит передача ST на сторону нужного принципала (сервиса).

Передача service ticket для идентификации на сервисе

Как я уже говорил, «сервисом» может был любая программа, предоставляющая возможность по запросу выполнять заранее известный функционал. Любой «сервис» может осуществлять коммуникации по любым протоколам. Один сервис «слушает» запросы только по HTTP, другой только по SMB, третий — по какому-то собственному проприетарному протоколу. Однако хотелось бы, чтобы можно было выполнять аутентификацию клиента как можно проще и унифицировано. Одним из способов такой унификации является использование Generic Security Service Application Program Interface (GSS-API, RFC 2743, RFC 4121). На самом деле в GSS-API описан просто набор общих функций, их параметров, а также некие общие заголовки передаваемых сообщений. Также описывается, какой именно результат должен быть достигнут от выполнения определённой функции с определёнными параметрами. То есть вроде «если выполнишь сначала такую, а потом вот такую функции, то аутентифицируешь клиента». Всё максимально просто. Собственно внутренняя реализация функций GSS-API остаётся за производителем операционной системы. В Windows реализация GSS-API известна под названием Security Support Provider Interface (SSPI). С помощью SSPI можно работать не только с Kerberos (и GSS-API изначально сделан с возможностью поддержки множества протоколов аутентификации). Можно даже писать свои собственные «провайдеры», обслуживающие аутентификацию по произвольным протоколам.

Однако вернёмся к работе с Kerberos с помощью SSPI. Сам по себе SSPI является неким набором стандартных функций, реализация же этих функций называется SSP (Security Support Provider). По сути, в операционной системе Windows SSP реализует функционал «стандартного клиента Kerberos», а также «стандартного сервера/сервиса Kerberos», и в Windows больше нет никаких других API, которые бы работали с Kerberos. Для «стандартного клиента» достаточно использовать только две функции: AcquireCredentialsHandle (реализация GSS_Acquire_cred из RFC 2743) и InitializeSecurityContext (реализация функции GSS_Init_sec_context из RFC 2743). Для «стандартного сервиса» достаточно также только двух функций: ранее упомянутой AcquireCredentialsHandle, и AcceptSecurityContext (реализация GSS_Accept_sec_context из RFC 2743). Функции InitializeSecurityContext и AcceptSecurityContext реализованы так, чтобы воспринимать выходную информацию друг друга. То есть на клиенте вызывается InitializeSecurityContext, затем сама эта функция внутри, если нужно, запрашивает TGT для нужного пользователя, затем запрашивается ST (service ticket) для нужного принципала, из-под которого выполняется нужный «сервис». Далее выходная информация из функции InitializeSecurityContext каким-то образом (безразлично каким конкретно, позже я покажу реальные реализации такой коммуникации) попадает на сторону «сервиса». Внутри «сервиса» выполняется вызов функции AcceptSecurityContext, в которую без изменений передаётся весь бинарный массив данных, ранее сформированный на клиенте с помощью InitializeSecurityContext. В качестве результата работы функции AcceptSecurityContext возвращаются

различные флаги. С помощью этих флагов можно определить, смогли ли внутренние алгоритмы AcceptSecurityContext идентифицировать клиента или нет. Также есть специальная функция QueryContextAttributes с помощью которой можно получить различные переменные «контекста», как то SPN того ST, который поступил на вход, имя клиента, которое существует в данном ST и так далее. В некоторых случаях может случиться так, что AcceptSecurityContext потребуется дополнительная информация со стороны клиента. В этом случае AcceptSecurityContext формирует свой выходной бинарный буфер, который также должен быть переслан обратно на сторону клиенте (безразлично каким именно способом). Ну и далее по кругу: InitializeSecurityContext — AcceptSecurityContext, до тех пор, пока AcceptSecurityContext не возвратит флаг ошибки или успешно идентифицирует клиента.

На самом деле все «выходные бинарные буфера» для обеих функций реализуют стандартное кодирование информации, принятое в GSS-API. Опишу заголовок данных при передаче сообщений Kerberos. Заранее упомяну, что обычно стандартные клиенты Kerberos использует SPNEGO (RFC4178), реализация которого есть в Windows в SSP (Security Support Provider) с именем «Negotiate». Однако SPNEGO это просто «надстройка», позволяющая выбрать при авторизации NTLM или Kerberos. В реальности можно вместо SPNEGO применять напрямую SSP Kerberos. Итак, в GSS-API все внутренние сообщения протокола сначала идентифицируются OID (Object Identifier) в формате ASN.1. Для Kerberos данный OID имеет значение «1.2.840.113 554.1.2.2». После OID идёт 2 байта типа внутреннего сообщения Kerberos. Эти байты могут иметь следующие значения:

- 0×0001 = KERB-AP-REQUEST
- 0×0002 = KERB-AP-REPLY
- 0×0003 = KERB-ERROR
- 0×0004 = KERB-TGT-REQUEST
- 0×0104 = KERB-TGT-REPLY

Последние два типа сообщений могут применяться только в том случае, когда «сервис» работает по схеме U2U (сервис не имеет имени и пароля пользователя и может обмениваться запросами только по предварительно согласованным сессионным ключам). Более подробно про U2U я напишу позже в статье.

За двумя байтами типа идут собственно данные Kerberos. То есть если тип GSS-API сообщения равен 0×0001 (KERB-AP-REQUEST), то далее хранится обычный AP-REQ. Если тип GSS-API сообщения равен 0×0003 (KERB-ERROR), то далее хранится обычное сообщение KRB-ERROR, кодированное в ASN.1. В случае использования GSS-API (а в Windows всегда используется GSS-API) внутри AP-REQ будет формироваться специальным образом сформированный authenticator. Особенным является поле checksum. Вместо использования стандартного алгоритма Kerberos для формирования checksum, в GSS-API применяется набор дополнительных флагов. Именно эти флаги регулируют, будет ли информация из AP-REQ использована для делегирования, ожидает ли клиент каких-то дополнительных сообщений (mutual authentication), информацию по channel binding. С полным перечнем этих флагов можно ознакомиться в RFC4121, item 4.1.1.

Разберём теперь более подробно параметры, которые необходимо передавать для каждой из упомянутых функций. Первая функция в нашем списке AcquireCredentialsHandle. Функция служит для последующей передачи в «security context» (контекст безопасности) информации по используемым реквизитам для входа (credentials). В качестве реквизитов для входа может использоваться как пустые значения, имя пользователя и пароль, или специальным образом кодированный идентификатор сертификата пользователя.

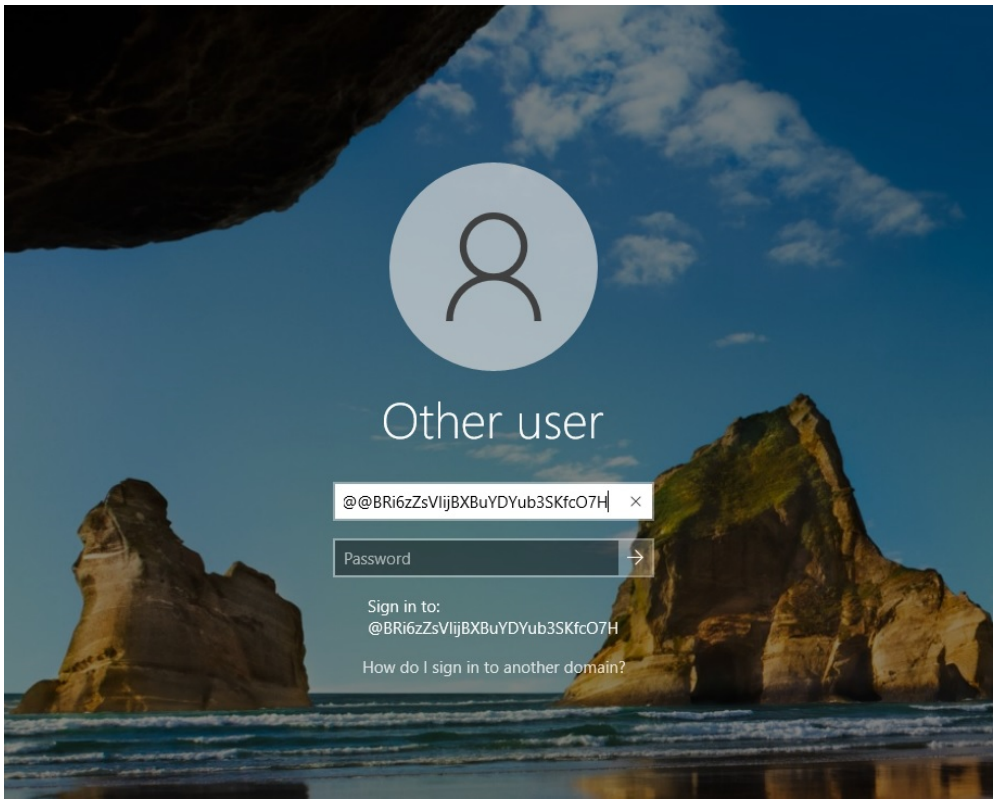
В случае пустых значений реквизиты для входа берутся из уже объявленных в текущей «logon session» (сессии входа в систему). Нужно заметить, что все данные, с которыми работают функции SSPI, а также все артефакты, которые они сохраняют на локальном компьютере, сохраняются в разрезе «logon sessions». На компьютере таких сессий может быть множество, для множества различных пользователей. Идентифицируются «logon sessions» по специальному идентификатору «Logon Unique IDentifier» (LUID), полный список существующих на компьютере сессий можно получить с помощью функции LsaEnumerateLogonSessions. Для того, чтобы в текущей logon session существовали реквизиты для входа необходимо, чтобы до этого они были явно в эту сессию добавлены. Это делается либо через вызов AcquireCredentialsHandle, либо с помощью вызова функции LsaLogonUser (которая, в свою очередь, неявно вызывает всё ту же AcquireCredentialsHandle). Если в текущей сессии уже есть реквизиты для входа, то можно реализовать «фантастический хак» — вызвать AcquireCredentialsHandle без указания пользователя и пароля и всё заработает.

Использование AcquireCredentialsHandle с указанием имени пользователя и пароля является тривиальным, и я пропущу объяснение этого варианта.

Гораздо более интересным является вариант указания реквизитов для входа с помощью идентификатора сертификата пользователя. Для реализации этого варианта необходимо, чтобы в хранилище сертификатов «Personal» (MY) текущего пользователя существовал сертификат, выданный центром сертификации текущего домена, в котором в качестве SAN (Subject Alternative Name) был бы указан UPN (User Principal Name) в формате «user@domain». Этот сертификат также должен иметь связанный закрытый ключ. Для формирования идентификатора сертификата, который можно передать в функцию AcquireCredentialsHandle, понадобится использовать функцию CredMarshalCredentialW, в которую должна быть передана структура типа CERT_CREDENTIAL_INFO. Основной информацией, которую содержит CERT_CREDENTIAL_INFO, является SHA-1 хэш необходимого нам сертификата. Такой хэш можно получить, например, с помощью функции CryptHashCertificate. Как результат вызова CredMarshalCredentialW мы получим строку, подобную этой: «@@BRi6zZsVIijBXbuYDYub3SKfcO7H». Пример программы, формирующей подобную строку из произвольного сертификата, можно найти по этой ссылке.

Эту строку можно передать в значение имени пользователя для функции AcquireCredentialsHandle. Если у используемого сертификата установлен PIN для доступа к закрытому ключу, то его также будет необходимо указать в значении «пароль пользователя». После указания подобного идентификатора сертификата ближайший вызов InitializeSecurityContext или AcceptSecurityContext будет формировать запрос на получение TGT с использованием PKINIT, а не с использованием стандартного способа шифрования на пароле. В результате использования сертификата для реквизитов входа можно получить сессию, в которой отсутствуют имя пользователя и пароль. Однако, как я уже писал ранее, Kerberos базируется на том, что для успешной коммуникации с такой сессией необходимо, чтобы была возможность использования пароля. Как же всё работает при использовании сертификатов (PKINIT)? В этом случае используется специальное расширение протокола, называемое «User-To-User» (U2U). При использовании этого расширения коммуникации производятся без помощи шифрования на пароле, а только с использованием согласованных сессионных ключей. Более подробно про это расширение я напишу позднее.

Как оказалось, результат вызова CredMarshalCredentialW можно передавать во многих случаях, когда требуется информация о реквизитах доступа. То есть, например, возможно указать строковый идентификатор сертификата при указании имени пользователя, из-под которого запуститься произвольный сервис. Или можно локально зайти в систему, вообще не зная имени пользователя и пароля. Однако, как я уже ранее писал, в этих случаях необходимо, чтобы используемый сертификат содержался в хранилище «Personal» для ***текущего*** пользователя. И в случае логина сервиса (здесь под «сервисом» понимается программа, запускаемая в оснастке «Services»), или выполнения первичного входа в систему, таким ***текущим*** пользователем будет SYSTEM. Для того, чтобы добавить произвольный сертификат в «Personal» для SYSTEM понадобится использовать любой «token stealer» и с его помощью запустить «mmc.exe». Там добавить оснастку «Certificates» и выполнить импорт сертификата. После этого будет доступен как бы функционал смарт-карт, но без использования самих смарт-карт, просто с использованием чистых сертификатов: вводите в поле «User name» строку, которую получили с помощью CredMarshalCredentialW и нажимаете Enter. Кстати, запуск сервисов без использования паролей полностью исключает использование атаки вида «Silver Ticket» так как даже если злоумышленник узнал пароль учётной записи для данного «сервиса» всё равно все коммуникации с этим сервисом будут проведены с помощью сессионного ключа и PAC всё равно будет сформирован на KDC.



Использование сертификата для интерактивного входа в систему

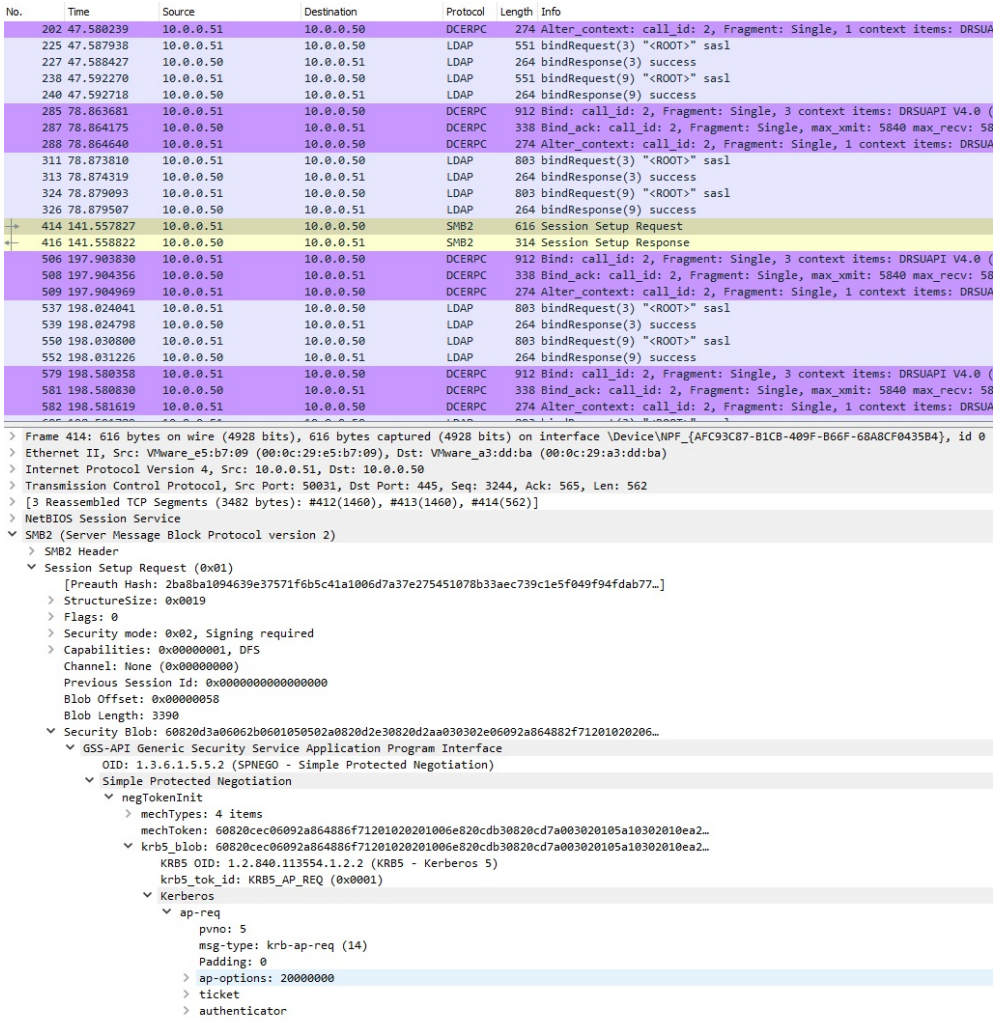
Перейдём к следующей функции: InitializeSecurityContext. Данная функция предназначена для создания первичного запроса к удалённому «сервису» и, если необходимо, передачи дополнительной информации, необходимой для окончательной идентификации клиента на удалённой системе. Как я уже писал ранее, все функции из SSPI работают с данными из текущей «logon session». Если в текущей сессии нет ранее полученного TGT, то первый вызов InitializeSecurityContext попытается этот TGT получить. Если же в сессии уже есть TGT, то будет просто сформирован запрос KDC-TGS-REQ, содержащий этот TGT и информацию о сервисе, ST (service ticket) для которого необходимо получить. После получения требуемого ST он сохраняется во внутренних структурах текущей «logon session». Если в дальнейшем любой программе, запускаемой в рамках той же «logon session», понадобится ST для того же сервиса, и этот ST будет иметь корректный временной интервал использования, то этот ST будет получен из «кэша», без запроса к KDC вообще. После получения требуемого ST функция InitializeSecurityContext формирует выходной бинарный буфер (его формат я уже ранее описывал) и ожидает, что этот буфер будет каким-то образом передан на сторону «сервиса». Примеры передачи подобного буфера представлены ниже.

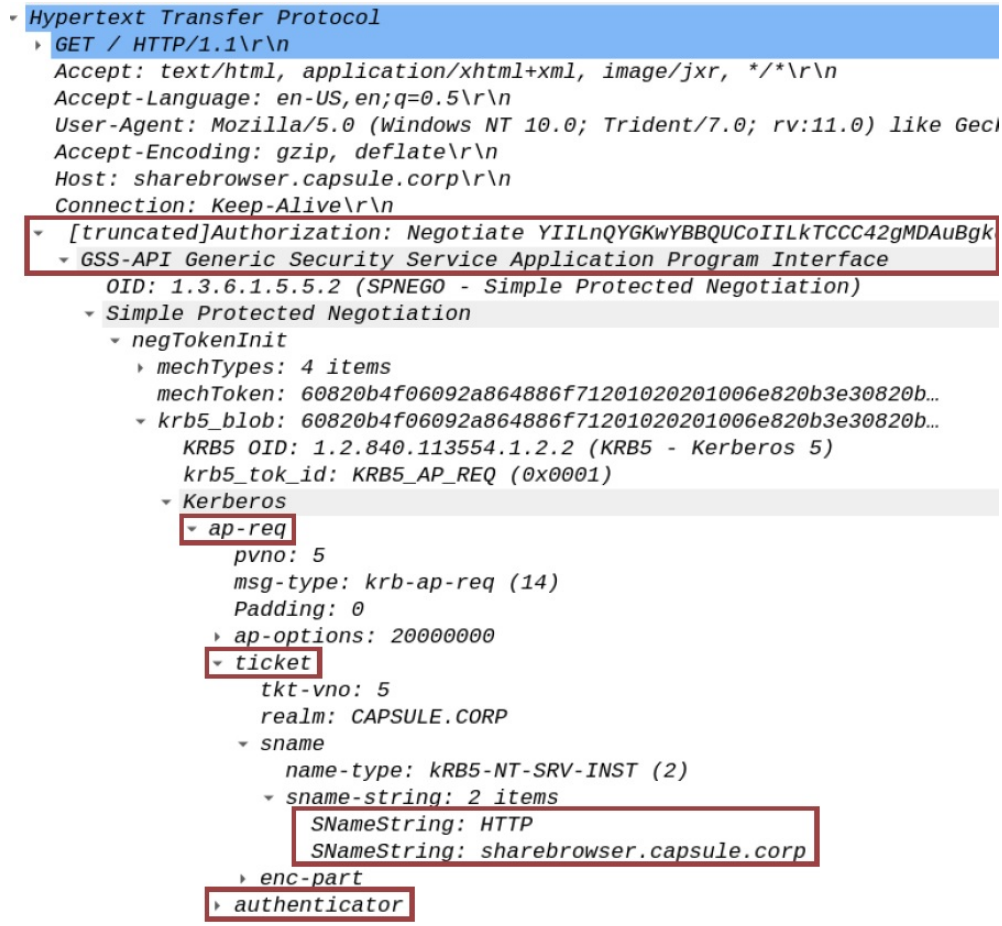
No.	Time	Source	Destination	Protocol	Length	Info
78	9.748617	10.0.0.50	10.0.0.51	LDAP	264	bindResponse(9) success
199	47.579323	10.0.0.51	10.0.0.50	DCERPC	660	Bind: call_id: 2, Fragment: Single, 3 context items: DRSUAPI V4.0 (3
201	47.579873	10.0.0.50	10.0.0.51	DCERPC	338	Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 584
202	47.580239	10.0.0.51	10.0.0.50	DCERPC	274	Alter_context: call_id: 2, Fragment: Single, 1 context items: DRSUAP
225	47.587938	10.0.0.51	10.0.0.50	LDAP	551	bindRequest(3) "<ROOT>" sasl
227	47.588427	10.0.0.50	10.0.0.51	LDAP	264	bindResponse(3) success
238	47.592270	10.0.0.51	10.0.0.50	LDAP	551	bindRequest(9) "<ROOT>" sasl
240	47.592718	10.0.0.50	10.0.0.51	LDAP	264	bindResponse(9) success

```
> Frame 63: 803 bytes on wire (6424 bits), 803 bytes captured (6424 bits) on interface \Device\NPF_{AFC93C87-B1CB-409F-B66F-68A8CF0435B4}, id 0
> Ethernet II, Src: VMware_e5:b7:09 (00:0c:29:e5:b7:09), Dst: VMware_a3:dd:ba (00:0c:29:a3:dd:ba)
> Internet Protocol Version 4, Src: 10.0.0.51, Dst: 10.0.0.50
> Transmission Control Protocol, Src Port: 49978, Dst Port: 389, Seq: 1811, Ack: 2705, Len: 749
> [2 Reassembled TCP Segments (2209 bytes): #62(1460), #63(749)]
v Lightweight Directory Access Protocol
  v LDAPMessage bindRequest(3) "<ROOT>" sasl
      messageID: 3
    v protocolOp: bindRequest (0)
      v bindRequest
          version: 3
          name:
        v authentication: sasl (3)
          v sasl
              mechanism: GSS-SPNEGO
              credentials: 6082087306062b0601050502a082086730820863a030302e06092a864882f71201020206…
            v GSS-API Generic Security Service Application Program Interface
                OID: 1.3.6.1.5.5.2 (SPNEGO - Simple Protected Negotiation)
              v Simple Protected Negotiation
                v negTokenInit
                  > mechTypes: 4 items
                    mechToken: 6082082506092a864886f71201020201006e8208143082081 0a003020105a10302010ea2…
                  v krb5_blob: 6082082506092a864886f71201020201006e8208143082081 0a003020105a10302010ea2…
                      KRB5 OID: 1.2.840.113554.1.2.2 (KRB5 - Kerberos 5)
                      krb5_tok_id: KRB5_AP_REQ (0x0001)
                    v Kerberos
                      v ap-req
                          pvno: 5
                          msg-type: krb-ap-req (14)
                          Padding: 0
                        > ap-options: 20000000
                        > ticket
                        > authenticator
    [Response In: 65]
```

No.	Time	Source	Destination	Protocol	Length	Info
240	47.592718	10.0.0.50	10.0.0.51	LDAP	264	bindResponse(9) success
285	78.863681	10.0.0.51	10.0.0.50	DCERPC	912	Bind: call_id: 2, Fragment: Single, 3 context items: DRSUAPI V4.0 (3
287	78.864175	10.0.0.50	10.0.0.51	DCERPC	338	Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 584
288	78.864640	10.0.0.51	10.0.0.50	DCERPC	274	Alter_context: call_id: 2, Fragment: Single, 1 context items: DRSUAP
311	78.873810	10.0.0.51	10.0.0.50	LDAP	803	bindRequest(3) "<ROOT>" sasl
313	78.874319	10.0.0.50	10.0.0.51	LDAP	264	bindResponse(3) success
324	78.879093	10.0.0.51	10.0.0.50	LDAP	803	bindRequest(9) "<ROOT>" sasl
326	78.879507	10.0.0.50	10.0.0.51	LDAP	264	bindResponse(9) success

```
> Frame 199: 660 bytes on wire (5280 bits), 660 bytes captured (5280 bits) on interface \Device\NPF_{AFC93C87-B1CB-409F-B66F-68A8CF0435B4}, id 0
> Ethernet II, Src: VMware_e5:b7:09 (00:0c:29:e5:b7:09), Dst: VMware_a3:dd:ba (00:0c:29:a3:dd:ba)
> Internet Protocol Version 4, Src: 10.0.0.51, Dst: 10.0.0.50
> Transmission Control Protocol, Src Port: 50006, Dst Port: 49669, Seq: 1461, Ack: 1, Len: 606
> [2 Reassembled TCP Segments (2066 bytes): #198(1460), #199(606)]
v Distributed Computing Environment / Remote Procedure Call (DCE/RPC) Bind, Fragment: Single, FragLen: 2066, Call: 2
    Version: 5
    Version (minor): 0
    Packet type: Bind (11)
  > Packet Flags: 0x07
  > Data Representation: 10000000 (Order: Little-endian, Char: ASCII, Float: IEEE)
    Frag Length: 2066
    Auth Length: 1898
    Call ID: 2
    Max Xmit Frag: 5840
    Max Recv Frag: 5840
    Assoc Group: 0x00000000
    Num Ctx Items: 3
  > Ctx Item[1]: Context ID:0, DRSUAPI, 32bit NDR
  > Ctx Item[2]: Context ID:1, DRSUAPI, 64bit NDR
  > Ctx Item[3]: Context ID:2, DRSUAPI, Bind Time Feature Negotiation
  v Auth Info: SPNEGO, Packet privacy, AuthContextId(0)
      Auth type: SPNEGO (9)
      Auth level: Packet privacy (6)
      Auth pad len: 0
      Auth Rsrvd: 0
      Auth Context ID: 0
    v GSS-API Generic Security Service Application Program Interface
        OID: 1.3.6.1.5.5.2 (SPNEGO - Simple Protected Negotiation)
      v Simple Protected Negotiation
        v negTokenInit
          > mechTypes: 4 items
            mechToken: 6e82071830820714a003020105a10302010ea2070305002000000a3820535618205313 0…
          v krb5_blob: 6e82071830820714a003020105a10302010ea2070305002000000a3820535618205313 0…
            v Kerberos
              v ap-req
                  pvno: 5
                  msg-type: krb-ap-req (14)
                  Padding: 0
                > ap-options: 20000000
                > ticket
                > authenticator
```


No. Time Source Destination Protocol Length Info
202 47.580239 10.0.0.51 10.0.0.50 DCERPC 274 Alter_context: call_id: 2, Fragment: Single, 1 context items: DRSUA
225 47.587938 10.0.0.51 10.0.0.50 LDAP 551 bindRequest(3) "<ROOT>" sasl
227 47.588427 10.0.0.50 10.0.0.51 LDAP 264 bindResponse(3) success
238 47.592270 10.0.0.51 10.0.0.50 LDAP 551 bindRequest(9) "<ROOT>" sasl
240 47.592718 10.0.0.50 10.0.0.51 LDAP 264 bindResponse(9) success
285 78.863681 10.0.0.51 10.0.0.50 DCERPC 912 Bind: call_id: 2, Fragment: Single, 3 context items: DRSUAPI V4.0 (
287 78.864175 10.0.0.50 10.0.0.51 DCERPC 338 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 58
288 78.864640 10.0.0.51 10.0.0.50 DCERPC 274 Alter_context: call_id: 2, Fragment: Single, 1 context items: DRSUA
311 78.873810 10.0.0.51 10.0.0.50 LDAP 803 bindRequest(3) "<ROOT>" sasl
313 78.874319 10.0.0.50 10.0.0.51 LDAP 264 bindResponse(3) success
324 78.879093 10.0.0.51 10.0.0.50 LDAP 803 bindRequest(9) "<ROOT>" sasl
326 78.879507 10.0.0.50 10.0.0.51 LDAP 264 bindResponse(9) success
414 141.557827 10.0.0.51 10.0.0.50 SMB2 616 Session Setup Request
416 141.558822 10.0.0.50 10.0.0.51 SMB2 314 Session Setup Response
506 197.903830 10.0.0.51 10.0.0.50 DCERPC 912 Bind: call_id: 2, Fragment: Single, 3 context items: DRSUAPI V4.0 (
508 197.904356 10.0.0.50 10.0.0.51 DCERPC 338 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 58
509 197.904969 10.0.0.51 10.0.0.50 DCERPC 274 Alter_context: call_id: 2, Fragment: Single, 1 context items: DRSUA
537 198.024041 10.0.0.51 10.0.0.50 LDAP 803 bindRequest(3) "<ROOT>" sasl
539 198.024798 10.0.0.50 10.0.0.51 LDAP 264 bindResponse(3) success
550 198.030800 10.0.0.51 10.0.0.50 LDAP 803 bindRequest(9) "<ROOT>" sasl
552 198.031226 10.0.0.50 10.0.0.51 LDAP 264 bindResponse(9) success
579 198.580358 10.0.0.51 10.0.0.50 DCERPC 912 Bind: call_id: 2, Fragment: Single, 3 context items: DRSUAPI V4.0 (
581 198.580830 10.0.0.50 10.0.0.51 DCERPC 338 Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 58
582 198.581619 10.0.0.51 10.0.0.50 DCERPC 274 Alter_context: call_id: 2, Fragment: Single, 1 context items: DRSUA
Frame 414: 616 bytes on wire (4928 bits), 616 bytes captured (4928 bits) on interface \Device\NPF_{AFC93C87-B1CB-409F-B66F-68A8CF0435B4}, id 0
Ethernet II, Src: VMware_e5:b7:09 (00:0c:29:e5:b7:09), Dst: VMware_a3:dd:ba (00:0c:29:a3:dd:ba)
Internet Protocol Version 4, Src: 10.0.0.51, Dst: 10.0.0.50
Transmission Control Protocol, Src Port: 50031, Dst Port: 445, Seq: 3244, Ack: 565, Len: 562
[3 Reassembled TCP Segments (3482 bytes): #412(1460), #413(1460), #414(562)]
NetBIOS Session Service
SMB2 (Server Message Block Protocol version 2)
SMB2 Header
Session Setup Request (0x01)
[Preauth Hash: 2ba8ba1094639e37571f6b5c41a1006d7a37e275451078b33aec739c1e5f049f94fdab77…]
StructureSize: 0x0019
Flags: 0
Security mode: 0x02, Signing required
Capabilities: 0x00000001, DFS
Channel: None (0x00000000)
Previous Session Id: 0x0000000000000000
Blob Offset: 0x00000058
Blob Length: 3390
Security Blob: 60820d3a06062b0601050502a0820d2e30820d2aa030302e06092a864882f71201020206…
GSS-API Generic Security Service Application Program Interface
OID: 1.3.6.1.5.5.2 (SPNEGO - Simple Protected Negotiation)
Simple Protected Negotiation
negTokenInit
mechTypes: 4 items
mechToken: 60820cec06092a864886f71201020201006e820cdb30820cd7a003020105a10302010ea2…
krb5_blob: 60820cec06092a864886f71201020201006e820cdb30820cd7a003020105a10302010ea2…
KRB5 OID: 1.2.840.113554.1.2.2 (KRB5 - Kerberos 5)
krb5_tok_id: KRB5_AP_REQ (0x0001)
Kerberos
ap-req
pvno: 5
msg-type: krb-ap-req (14)
Padding: 0
ap-options: 20000000
ticket
authenticator



Последний скриншот взят из этой презентации

Если при первом запуске для InitializeSecurityContext был использован флаг ISC_REQ_MUTUAL_AUTH, то функция будет ожидать ответный буфер, сформированный на стороне «сервиса» с помощью AcceptSecurityContext. Со своей стороны я рекомендую всегда использовать флаг ISC_REQ_MUTUAL_AUTH так как сервис может вернуть какую-то ошибку, которую на клиенте можно обработать. Например, «сервис» может вернуть ошибку с кодом, указывающим на то, что требуется прохождение идентификации на основе сессионного ключа (U2U). И если клиент не ожидает входящего сообщения, то коммуникация с приходом этой ошибки будет полностью прекращена. Значение всех используемых в SSPI флагов можно найти по этой ссылке.

Работа функции AcceptSecurityContext в начале аналогична работе InitializeSecurityContext: функция AcceptSecurityContext также может получить TGT для «сервисной» части, если это будет нужно. В этой функции также можно инициализировать контекст безопасности с использованием явно заданных имени пользователя, пароля или сертификата. Однако в отличие от InitializeSecurityContext функция AcceptSecurityContext просто ждёт входящих сообщений. После получения входящего сообщения (в виде бинарного буфера, сформированного на клиенте с помощью InitializeSecurityContext) функция AcceptSecurityContext может либо вернуть какой-то свой бинарный буфер, который подлежит отправке на сторону клиента, либо завершается с кодом, соответствующим успешному прохождению идентификации для удалённого клиента. В последнем случае «сервис» может, например, выделить отдельный процесс и в нём вызвать функцию ImpersonateSecurityContext — в этом случае этот новый процесс будет использовать «access token» (токен безопасности) в котором будут представлены все данные по идентификационной информации удалённого клиента (все его группы, привилегии и так далее). Важно заметить, что если функция AcceptSecurityContext смогла успешно идентифицировать удалённого клиента, то это автоматически приводит к тому, что на локальном для «сервиса» компьютере появляется новая «logon session» для удалённого пользователя. Это необходимо, в частности, для того, чтобы «сервис» в дальнейшем мог организовать «делегирование» идентификационной информации (об этом позже в статье). В этой новой «logon session» по умолчанию будет отсутствовать какая-либо информация по реквизитам доступа: там не будет TGT, не будет сохранённого имени пользователя и пароля. Всё, что будет в этой новой сессии это «access token», который можно использовать, например, для проверки прохождения авторизации на локальных ресурсах. Таким образом, в общем случае функция AcceptSecurityContext в качестве своей конечной выходной информация позволяет создавать на локальной системе новую «logon session», а также новый «access token», содержащий всю идентификационную информацию для удалённого клиента. Необходимо сказать, что AcceptSecurityContext создаёт такой «access token» используя исключительно только ту идентификационную информацию, которая поступила этой функции на вход. То есть AcceptSecurityContext самостоятельно не выполняет никаких обращений к KDC, за исключением вызова функций проверки «подписей» внутри PAC. Ещё одной особенностью AcceptSecurityContext является тот факт, что у этой функции нет никакого параметра, в который бы передавалось имя того «сервиса», от имени которого эта функция в настоящий момент работает. То есть этой функции нет разницы работает она от имени сервиса «host/server», или «cifs/server» — всё, что ей нужно это реквизиты входа (credentials) текущего «сервиса». Вот здесь можно почитать про то, как человек переписывался с Microsoft по этому вопросу, и ему подтвердили, что фактически имя сервиса не является каким-то влияющим на аутентификацию параметром.

Дополнительной особенностью «сервисного контекста» в SSPI является то, что в этом контексте может существовать одновременно несколько credentials. То есть на самом деле один «сервис» может корректно обрабатывать запросы, направляемые в адрес нескольких различных пользователей/сервисов. Например, если у «user_1» сконфигурирован SPN_1, а у user_2 сконфигурирован SPN_2, то можно сделать так, что «сервисный контекст» (посредством функции AcceptSecurityContext) сможет корректно обработать входящие service tickets как для SPN_1, так и для SPN_2. Реализуется подобный функционал с помощью вызовов функции LsaCallAuthenticationPackage с типами сообщений KerbAddExtraCredentialsMessage или KerbAddExtraCredentialsExMessage. Добавление дополнительных идентификационных данных будет работать только в текущей logon session, в других сессиях дополнительные идентификационные данные будут отсутствовать.

Теперь, когда я описал основные функции, которые создают изменения в «контексте безопасности» (security context) необходимо также рассказать о функциях получения информации из «контекста безопасности». Таких функций всего две: QueryContextAttributes и LsaCallAuthenticationPackage. Описание первой функции можно найти здесь. С её помощью можно получить крайне ограниченную информацию из контекста безопасности. Например, на стороне «сервиса» после успешной идентификации с помощью AcceptSecurityContext функция QueryContextAttributes может получить непосредственное значение «access token», получить значение SPN из ST, или имя пользователя из ST. Функция LsaCallAuthenticationPackage является гораздо более интересной. Данная функция служит неким общим интерфейсом для вызова внутреннего функционала произвольного SSP. Реализуется подобная универсальность посредством существования у каждого «security provider» собственных наборов входных и выходных параметров, которые приводятся к нейтральным общим типам и передаются в LsaCallAuthenticationPackage. Перечень входных параметров для Kerberos можно найти по этой ссылке. Однако в официальном описании отсутствуют описания типов входных и выходных структур, передаваемых вместе с каждым из типов сообщений. Ниже я привожу собственную таблицу таких соответствий. Для более прозрачной работы я реализовал специализированные функции и типы внутри библиотеки XKERB.

Тип	Тип структуры входных данных	Тип структуры выходных
KerbDebugRequestMessage	?	?
KerbQueryTicketCacheMessage	KERB_QUERY_TKT_CACHE_REQUEST	KERB_QUERY_TKT_CAC
KerbChangeMachinePasswordMessage	?	?
KerbVerifyPacMessage	?	?
KerbRetrieveTicketMessage	KERB_RETRIEVE_TKT_REQUEST	KERB_RETRIEVE_TKT_R
KerbUpdateAddressesMessage	?	?
KerbPurgeTicketCacheMessage	KERB_PURGE_TKT_CACHE_REQUEST	
KerbChangePasswordMessage	KERB_CHANGEPASSWORD_REQUEST	
KerbRetrieveEncodedTicketMessage	KERB_RETRIEVE_TKT_REQUEST	KERB_RETRIEVE_TKT_R
KerbDecryptDataMessage	KERB_DECRYPT_REQUEST	KERB_DECRYPT_RESPC
KerbAddBindingCacheEntryMessage	KERB_ADD_BINDING_CACHE_ENTRY_REQUEST	
KerbSetPasswordMessage	KERB_SETPASSWORD_REQUEST	
KerbSetPasswordExMessage	KERB_SETPASSWORD_EX_REQUEST	
KerbAddExtraCredentialsMessage	KERB_ADD_CREDENTIALS_REQUEST	
KerbVerifyCredentialsMessage	?	?
KerbQueryTicketCacheExMessage	KERB_QUERY_TKT_CACHE_REQUEST	KERB_QUERY_TKT_CAC
KerbPurgeTicketCacheExMessage	KERB_PURGE_TKT_CACHE_EX_REQUEST	
KerbRefreshSmartcardCredentialsMessage	KERB_REFRESH_SCCRED_REQUEST	

KerbQuerySupplementalCredentialsMessage	?	?
KerbTransferCredentialsMessage	KERB_TRANSFER_CRED_REQUEST	
KerbQueryTicketCacheEx2Message	KERB_QUERY_TKT_CACHE_REQUEST	KERB_QUERY_TKT_CAC
KerbSubmitTicketMessage	KERB_SUBMIT_TKT_REQUEST	
KerbAddExtraCredentialsExMessage	KERB_ADD_CREDENTIALS_REQUEST_EX	
KerbQueryKdcProxyCacheMessage	KERB_QUERY_KDC_PROXY_CACHE_REQUEST	KERB_QUERY_KDC_PRC
KerbPurgeKdcProxyCacheMessage	KERB_PURGE_KDC_PROXY_CACHE_REQUEST	KERB_PURGE_KDC_PRC
KerbQueryTicketCacheEx3Message	KERB_QUERY_TKT_CACHE_REQUEST	KERB_QUERY_TKT_CAC
KerbCleanupMachinePkinitCredsMessage	KERB_CLEANUP_MACHINE_PKINIT_CREDS_REQUEST	
KerbAddBindingCacheEntryExMessage	KERB_ADD_BINDING_CACHE_ENTRY_EX_REQUEST	
KerbQueryBindingCacheMessage	KERB_QUERY_BINDING_CACHE_REQUEST	KERB_QUERY_BINDING_
KerbPurgeBindingCacheMessage	KERB_PURGE_BINDING_CACHE_REQUEST	
KerbPinKdcMessage	KERB_PIN_KDC_REQUEST	
KerbUnpinAllKdcsMessage	KERB_UNPIN_ALL_KDCS_REQUEST	
KerbQueryDomainExtendedPoliciesMessage	KERB_QUERY_DOMAIN_EXTENDED_POLICIES_REQUEST	KERB_QUERY_DOMAIN_
KerbQueryS4U2ProxyCacheMessage	KERB_QUERY_S4U2PROXY_CACHE_REQUEST	KERB_QUERY_S4U2PRC
KerbRetrieveKeyTabMessage	KERB_RETRIEVE_KEY_TAB_REQUEST	KERB_RETRIEVE_KEY_T
KerbRefreshPolicyMessage	KERB_REFRESH_POLICY_REQUEST	KERB_REFRESH_POLICY
KerbPrintCloudKerberosDebugMessage	KERB_CLOUD_KERBEROS_DEBUG_REQUEST	KERB_CLOUD_KERBERC

Как можно заметить, с помощью LsaCallAuthenticationPackage можно как просто получать информацию, так и запрашивать действия со стороны «security provider». Для автоматизации определённых вызовов LsaCallAuthenticationPackage я написал вспомогательный код, который можно найти здесь.

Также здесь можно упомянуть, что в Windows существует стандартная утилита «klist», с помощью которой можно посмотреть все существующие в текущей «logon session» TGT и service tickets. С помощью LsaCallAuthenticationPackage я реализовал аналог функциональности программы «klist», выводящий информацию по кэшированным тикетам Kerberos, код приведён здесь.

Ещё одной, играющей вспомогательную роль, является функция LsaLogonUser. Эта функция также запрашивает новый TGT для пользователя по его реквизитам для входа. Напомню, что в этой функции также можно использовать текстовое представление идентификатора сертификата. Однако в отличие от ранее рассмотренных функций LsaLogonUser после успешного получения TGT автоматически создаёт новую «logon session». Как я ранее писал, такой функциональностью из ранее рассмотренных функций обладает только AcceptSecurityContext. Так как все действия в SSPI так или иначе привязаны к «logon session», то лично я считаю, что если необходимо в программе реализовать какой-то «сервис» имея предопределённые реквизиты для входа, то крайне полезным будет сначала явно сделать «logon session», и уже потом в рамках этой новой сессии вызывать функции InitializeSecurityContext/AcceptSecurityContext. С использованием данного подхода я реализовал некий «универсальный клиент-сервисный код»: в одной функции вызываются как клиентские, так и сервисные функции. Это позволяет протестировать клиент-сервисное взаимодействие без использования каких-либо коммуникаций. Данный код находится здесь.

На этом я заканчиваю рассмотрение базового функционала, и перехожу ещё к одной большой теме — делегированию в Kerberos.

Делегирование

Под «делегированием» в Kerberos понимается процесс, когда клиент каким-либо образом передаёт «сервису» возможность действовать, используя идентификационную информацию клиента. Это бывает необходимо в тех случаях, когда клиент поручает «сервису» выполнить операцию, которая, в свою очередь, требует обращения к другим «сервисам». В этом случае первичному «сервису» будет необходимо получить возможность самостоятельно получать Service Ticket (ST) от имени клиента к другим «сервисам». Конечно, можно было бы как-то организовать обратную коммуникацию с клиентом, запрос у него ST для другого «сервиса», передача этого ST обратно и так далее. Однако технически это зачастую сложно реализуемо. Например, если «сервис» является HTTP сервисом, то у него очень ограниченная возможность обратной коммуникации с клиентом по своей инициативе. Поэтому Kerberos предоставляет функционал, позволяющий делегировать идентификационную информацию клиентов на определённые «сервисы».

Существуют следующие виды делегирования:

1. Неограниченное делегирование;
2. Ограниченное делегирование с использование только протокола Kerberos;
3. Ограниченное делегирование с использование произвольного протокола;

Перед описание каждого вида делегирования подробно приведу одно общее замечание. Чтобы «сервис» мог кешировать (и в дальнейшем использовать) TGT от других пользователей данный «сервис» должен исполняться из-под пользователя, у которого активна привилегия SeEnableDelegationPrivilege (SE_ENABLE_DELEGATION_NAME).

Неограниченное делегирование

В случае неограниченного делегирования клиент передаёт на сервис свой TGT вместе с его текущим сессионным ключом. Имея эту информацию «сервис» в дальнейшем может самостоятельно запросить Service Ticket (ST) к любому существующему «сервису». Коммуникации никак не ограничены.

Ограниченное делегирование с использованием только Kerberos

В случае ограниченного делегирования вместо TGT на сторону «сервиса» передаётся только ST. Далее «сервис» может, используя расширение S4U2Proxy, запросить от имени пользователя ST к другому «сервису». Набор «сервисов», к которым данный «сервис» может получить ST от имени пользователя, заранее определён и ограничен.

Несмотря на то, что механизм ограниченного делегирования с использованием только Kerberos реализован в Windows уже очень давно, и, вроде как, должен быть полностью отлажен у меня в тестовых сервисах не получилось реализовать делегирование с использованием этого режима. После аутентификации клиента на сервисе 1 в сессии возникал соответствующий service ticket от этого клиента к сервису 1. И этот service ticket даже имел установленный FORWARDABLE флаг. Однако если я пытался из этой сессии выполнить запрос к сервису 2, то функции SSPI не включали уже имеющийся service ticket и вместо этого пытались заново запросить TGT для клиента с использованием S4U2Self. В ответ формировался TGT, однако без флага FORWARDABLE. И, как результат, запрос на получение service ticket для сервиса 2 отклонялся.

Ограниченное делегирование с использованием произвольного протокола

Здесь клиент может присылать на сервис только service ticket, однако «сервис» имеет дополнительную возможность самостоятельно, вообще без какого-либо участия клиента запрашивать его идентификационную информацию. Повторюсь, что в случае «делегирование без ограничения протоколов» пользователь всё также может передать ST с на сторону «сервиса», эта функциональность в данном виде делегирования осталась без изменений.

Для реализации запроса «сервисом» идентификационной информации для произвольного клиента было реализовано расширение Service for User to Self (S4U2self). Представим, что есть какой-то «сервис», который выполняет идентификацию пользователей по биометрии, скажем на основании анализа лица пользователя. И для этого «сервис» использует специфическое оборудование, специальные вызовы драйверов и так далее. База данных лиц пользователей также будет какой-то специализированной. Также предположим, что этот «сервис» должен выполнять аутентификацию пользователей домена, и использовать для этого Kerberos. В случае подобной задачи внутри «сервиса» будет нужна дополнительная база, сопоставляющая имена пользователей домена с их идентификационными данными, специфичными для данного «сервиса» (в нашем случае — с изображениями их лиц). Таким образом, после успешной идентификации пользователя по его лицу «сервис» теперь может получить имя пользователя в домене. Далее, используя расширение S4U2Self, данный «сервис» сможет запросить аутентификационную информацию пользователя по протоколу Kerberos зная только его имя и далее получить возможность аутентификации данного пользователя на других «сервисах» (логин на рабочую станцию, доступ к файлам по сети и так далее).

На самом деле использовать S4U2Self может абсолютно любой пользователь, который уже прошёл идентификацию в домене. То есть если у пользователя есть TGT, то он может запросить идентификационную информацию по любому другому пользователю, машине или даже «сервису» (по имени «сервиса»). Нет никаких ограничений на то, от имени кого будут приходить запросы S4U2Self, запросы может делать как полноценный «сервис», так и обычный пользователь.

Расширение S4U2Self реализуется с помощью всего двух дополнительных структур: PA-FOR-USER и PA-S4U-X509-USER. Использование этих двух структур является взаимоисключающим (можно использовать только один из двух типов в одном запросе). При использовании PA-FOR-USER можно запрашивать информацию только на основе имени пользователя.

При использовании PA-S4U-X509-USER можно запрашивать информацию как на основе имени пользователя, так и на основе X.509 сертификата пользователя. В MS-SFU также упоминается, что если текущий домен поддерживает схемы шифрования с использованием AES-128 и выше, то рекомендуется всегда использовать PA-S4U-X509-USER.

Реализация SSP Kerberos в Windows поддерживает использование S4U2Self, но только опосредованно, через вызов LsaLogonUser с параметрами KERB_S4U_LOGON или KERB_CERTIFICATE_S4U_LOGON. На основании использования этой функция я создал некий «whoami на S4U», позволяющий выводить всю информацию любого пользователя — как то его группы, SID, набор привилегий. Код данного проекта доступ по этой ссылке.

Использование S4U2Self позволяет получить идентификационную информацию для любого пользователя, однако это позволяет идентифицировать пользователя только локально на той машине, на которой запущен «сервис». Для того, чтобы реализовать возможность делегирования идентификационных данных пользователей на других «сервисах» было реализовано другое расширение Kerberos — S4U2Proxy.

Расширение S4U2Proxy позволяет запрашивать у KDC service ticket от имени любого пользователя к ограниченному набору сервисов. Внутри расширение S4U2Proxy реализовано просто в виде двух дополнительных флагов: флага «resource-based constrained delegation» внутри PA-PAC-OPTIONS, и флага «cname-in-addl-tkt» внутри KDC_OPTIONS. Использование флага CNAME-IN-ADDL-TKT позволяет запросить ST для того пользователя, имя которого содержится в «additional tickets». Использование флага «resource-based constrained delegation» позволяет использовать возможности RBCD (Resource-Based Constrained Delegation).

Как ни странно, несмотря на официальную документацию Microsoft (MS-SFU), флаг CNAME-IN-ADDL-TKT на самом деле не используется. То есть ни одна функция SSPI не выставляет данный флаг и не учитывает его при приёме сообщений. Как минимум так было у меня во всех возможных проводимых тестах. Вместо этого, похоже, имя клиента автоматически берётся из additional ticket если это поле содержит какое-то значение.

Использование RBCD позволяет более тонко конфигурировать получение service tickets для каждого из «сервисов». При использовании RBCD каждый пользователь, из-под которого запускается какой-либо «сервис», может сконфигурировать свой список других пользователей, которым доступна возможность делегирования на этот «сервис». То есть в атрибуте msDS-AllowedToActOnBehalfOfOtherIdentity для данного пользователя может содержаться полноценный security descriptor (SD), в котором будут прописаны полноценные Access Control Lists (ACLs). Представим, что клиент сформировал запрос для «сервиса 1». В процессе деятельности «сервис 1» должен выполнить какие-то действия на «сервис 2» используя аутентификационную информацию для клиента. При использовании RBCD «сервис 2» должен иметь в msDS-AllowedToActOnBehalfOfOtherIdentity такой security descriptor, который бы позволял доступ для пользователя, из-под которого запускается «сервис 1». Также необходимо понимать крайне важную особенность RBCD: если для какого-то пользователя сконфигурирован SD, который позволяет выполнить делегирование для «сервис 1», то режим делегирования для «сервис 1» не учитывается. То есть если «сервис 1» сконфигурирован так, чтобы не позволять делегирование (режим «Do not trust this user for delegation»), то он всё равно сможет выполнить делегирование к «сервис 2», если у «сервис 2» сконфигурирован security descriptor, позволяющий доступ для «сервис 1». Также KDC не учитывает ограничения делегирования по именам. То есть если для «сервис 1» задано ограниченное делегирование, и «сервис 2» не присутствует в списке, то делегирование на «сервис 2» всё равно будет выполнено, если на «сервис 2» задан корректный security descriptor, позволяющий доступ для «сервис 1».

User-To-User (U2U)

Как уже было сказано, протокол Kerberos использует шифрование на заранее известных ключах, генерируемых на основе текстовых паролей. Однако существуют варианты, когда пользователь будет запускать какой-либо «сервис», не обладая знанием о собственном пароле. Это может быть, например, в случае использования X.509 сертификата для первичной аутентификации. В этом случае необходимо иметь возможность так или иначе обмениваться аутентификационной информацией между клиентами и подобным «сервисом». Для реализации подобного функционала была придумана схема U2U.

При использовании U2U работа с «сервисом» для клиента начинается с посылки стандартного сообщения AP-REQ. Однако, так как «сервис» не имеет информации о собственном пароле, то «сервис» в ответ формирует ошибку с кодом KRB_AP_ERR_USER_TO_USER_REQUIRED (0×45). Для того, чтобы клиент смог получить эту ошибку на «сервисе» должен быть использован флаг ASC_REQ_EXTENDED_ERROR, а на клиенте должен быть использован флаг ISC_REQ_MUTUAL_AUTH чтобы клиент ожидал какой-то информации со стороны «сервиса». Можно также явно задать флаги ISC_REQ_USE_SESSION_KEY и ASC_REQ_USE_SESSION_KEY, но в этом случае общая система теряет универсальность: клиент сразу начинает с запроса TGT для «сервиса».

Итак, после получения ошибки клиент понимает, что необходимо реализовать общение с использованием не постоянных, а сессионных ключей. Для этого клиент сперва формирует запрос на «сервис» для получения TGT этого сервиса. Данный запрос формируется на уровне GSS-API, и представляет собой, по сути, специальный Object Identificator (OID). В ответ «сервис» возвращает свой TGT. Передача TGT в данном случае условно безопасна, так как вместе с TGT не передаётся его сессионный ключ. После получения этого TGT клиент формирует специальный запрос к KDC на получение service ticket, в котором устанавливается флаг ENC-TKT-IN-SKEY, а также в поле additional tickets добавляется TGT, полученный со стороны «сервиса». В свою очередь KDC расшифровывает TGT из additional ticket, получает оттуда session key, а затем формирует новый service ticket, шифруя его уже на этом session key. После получения нового service ticket (ST) клиент далее может переслать этот ST заново необходимому «сервису». Теперь «сервис» может расшифровать этот полученный ST и провести аутентификацию клиента стандартным образом.

Тестирование Kerberos

Теперь, когда я описал все основные возможности стандартных клиента и серверной части Kerberos в Windows, можно перейти к описанию схемы тестирования этого функционала. Так как в Windows только в SSPI реализована возможность работы с Kerberos, то естественно, что для тестирования будут использоваться только стандартные функции SSPI. Например, вот здесь приведены ссылки на тестовые примеры по SSPI от Microsoft. В этих примерах существует однозначное разделение на программу, выполняющую клиентскую часть, и программу, которая реализует функции «сервиса». Однако всё можно упростить, если убрать передачу между «клиентом» и «сервисом» по какому-либо стороннему каналу (вроде pipe, как в примерах от Microsoft). Вместо этого достаточно, чтобы выходной массив из функции InitializeSecurityContext был передан на вход функции AcceptSecurityContext. Для реализации полноценных «клиента» и «сервиса» на самом деле достаточно использовать двух обычных пользователей домена, у одного из которых будет непустой атрибут servicePrincipalNames (то есть будут SPNs). Изменяя параметры делегирования для «сервисного» пользователя, можно изучать различные виды этого делегирования, а также иметь возможность проконтролировать каждый из параметров как на стороне «клиента», так и «сервиса». Можно инициализировать «сервис» через сертификат (PKINIT) и получить сервисный контекст, в котором будут отсутствовать данные о логине и пароле, а обмен с таким «сервисом» будет возможен только через U2U. Можно добавлять дополнительные credentials в «сервисный» контекст и получить возможность корректно принимать «не свои» билеты: то есть если у нас в тестовом окружении будут два «сервисных» пользователя, и мы добавим credentials от второго пользователя в контекст первого, то теперь сервисный контекст первого пользователя сможет корректно обработать все AP-REQ к SPNs от второго пользователя, абсолютно прозрачно. Можно добавлять tickets в «клиентский» или «сервисный» контекст (Push The Ticket, PTT). Можно получать какие-то дополнительные данные из любого контекста, и так далее.

Для более удобной работы со всем этим функционалом я реализовал библиотеку XKERB, а также набор тестовых функций, позволяющий тестировать произвольный функционал SSPI. Библиотека написана на чистом C++ с использованием функционала последнего стандарта C++ 23. При работе применяются «умные указатели» и, как следствие, пользователю библиотеки нет необходимости заботиться о менеджменте памяти: все необходимые функции будут вызваны в нужное время и память будет корректно освобождена. Также в библиотеке применяется «Aggregate initialization», позволяющая передавать в функции параметры по их имени, а также прозрачно использовать опциональные параметры. Реализация библиотеки находится по этой ссылке.