

Posts By SpecterOps Team Members

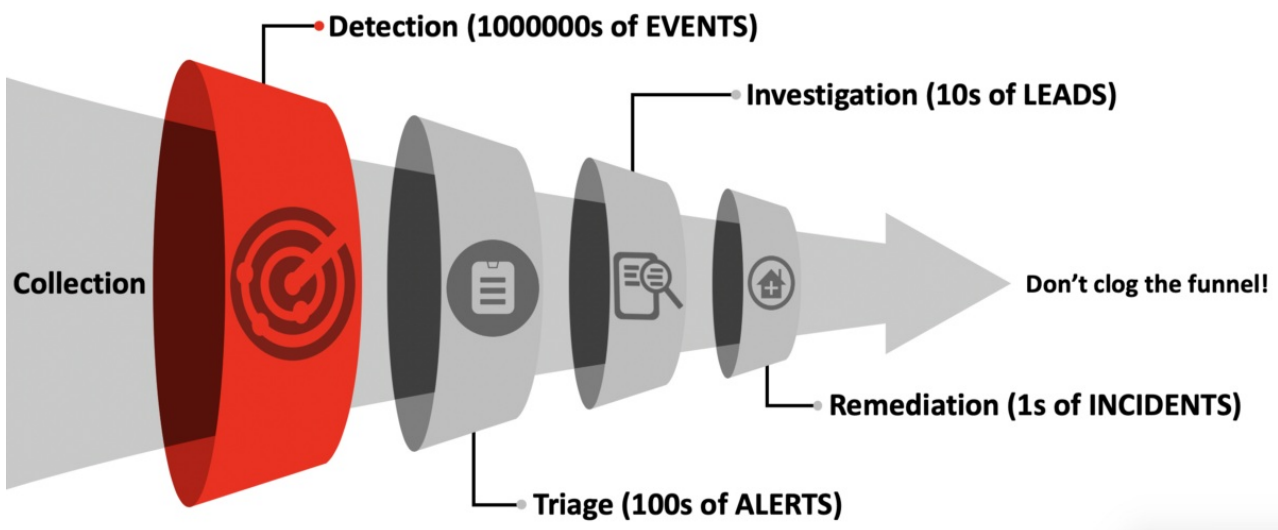
posts.specterops.io/capability-abstraction-fbeaeeb26384

February 6,
2020

Capability Abstraction



In 2020, the SpecterOps detection team members are making an effort to relate the concepts of their blog posts to the Funnel of Fidelity. The Funnel of Fidelity is a model that SpecterOps uses to contextualizes the phases that must occur to successfully detect and remediate an attack. Our hope is this helps readers understand where the post fits into the context of their overall detection program. This specific post is focused on a strategy that can be applied to the Detection phase of the funnel.



Introduction

This is the first of a multipart blog series by the SpecterOps detection team. The goal of this series is to introduce and discuss foundational detection engineering concepts. To make these concepts as consumable as possible, we are focusing the entire series around Kerberoasting. Focusing on this technique allows readers to focus on the strategies presented in each article instead of worrying about the details of the technique itself. The focus of this post is a concept we call “capability abstraction.” The idea is that an attacker’s tools are merely an abstraction of their attack capabilities, and detection engineers must understand how to evaluate abstraction while building detection logic.

What is Abstraction?

You may be wondering what I mean when I say “abstraction.” In the context of this article, abstraction is a concept by which implementation details are hidden from/automated for the user. Abstraction typically manifests itself in layers, called abstraction layers, that systematically hide complexity with each layer providing a more superficial interface than the last. Abstraction is an important design element because technology products often target a user base that fundamentally does not understand how the products work but can still benefit from the use of the technology. Ultimately, one of the main goals of abstraction is to hide the complex inner workings of something to make it more approachable to a wider audience.

A second reason for abstraction is to streamline interaction with technology. In information technology, we often say, “if you have to do it more than once, automate it.” Similarly, attackers may want to streamline their attack procedures as much as possible, within their acceptable OPSEC constraints. We see this concept expressed in the tools that adversaries use. Each tool provides some level of abstraction in the form of a common interface for operators to use.

I believe that if we can peel back the abstraction layers to understand the fundamental attack, we can make more informed decisions regarding the best ways to detect an attack technique.

What is Kerberoasting?

I want to spend some time talking about Kerberoasting before getting too far down the capability abstraction rabbit hole. In a macro sense, Kerberoasting is an attack technique that allows attackers to convert Kerberos ticket-granting service tickets (TGS tickets) into passwords. At a micro (more detailed) level, Kerberos authentication uses a service principal name (SPN) to uniquely identify a service instance. Each SPN associates a service instance (think web server, SQL Server, etc.) to a service logon account. To authenticate to a remote system, a user must request a TGS ticket for the target service which is identified by the SPN. A portion of the TGS ticket is encrypted using the password hash of the service logon account that is associated with the target SPN. To Kerberoast, attackers extract that encrypted portion and attempt to brute force the decryption of that section until they are successful. Once they successfully decrypt the ticket, they know the service logon account’s password, can assume the identity of that account, and log in to any systems that the account has access to.

If you are interested in digging deeper into Kerberoasting before continuing, I recommend checking out Tim Medin’s [Attacking Kerberos: Kicking the Guard Dog of Hades](#) presentation where he introduced the concept (the slides can be found [here](#)). Additionally, harmj0y has written a few blog posts on the topic, specifically his [Kerberoasting without Mimikatz](#) post where he links to numerous other sources on the topic.

Abstraction of Kerberoasting

Throughout this discussion, this post leverages a visual graphic referred to as an “abstraction map” that aims to help readers follow along with the relationships between the abstraction layers as they are discussed. The idea is to reveal and track the abstraction layers as we discover them. At this point, we haven’t explored Kerberoasting at all, so our initial abstraction map below is empty.

Let’s get started!

Tools

It is important to consider that attack tools represent the most superficial layer of abstraction; however, a quick review of publicly shared detection logic reveals that defenders often focus detection engineering efforts on tool-specific signatures. These superficial detections are not inherently bad but are prone to blind spots (false negatives) without further contextual data. The goal of evaluating abstraction’s role in an attack technique is to inform detection engineering efforts. This process can help avoid false negatives, learn about potential telemetry needs, understand new attack implementations, and determine when the detection approach sufficiently covers our target technique.

Invoke-Kerberoast

Invoke-Kerberoast is a PowerShell advanced function that allows an attacker to request a Kerberos service ticket for a target account. In addition to requesting the service ticket, Invoke-Kerberoast also extracts the encrypted portion of the ticket and returns it in a format that can be cracked offline using popular password cracking tools like John the Ripper and Hashcat. Below is an example of executing Invoke-Kerberoast to request a ticket in a test domain.

```
PS C:\PowerSploit\PowerSploit-dev> Invoke-Kerberoast -Identity ironman

SamAccountName      : ironman
DistinguishedName   : CN=Tony Stark,CN=Users,DC=marvel,DC=local
ServicePrincipalName : mr3000/marvel.local
TicketByteHexStream :
Hash                : $krb5tgs$23$*ironman$marvel.local$mr3000/marvel.local*$A4717016BAB9ECFECBF102A77D395407$08615FBC
```

Invoke-Kerberoast execution

At this point, the perspective of kerberoasting is limited to just a single tool (Invoke-Kerberoast). Let’s take a second to update the abstraction map with Invoke-Kerberoast:

T1208 - Kerberoasting

Tool


PowerShell
Invoke-Kerberoast

When considering the current limited perspective of the problem, it makes sense why it may be common for detection engineers to create a simple detection for Invoke-Kerberoast and call it a day. Detections built for Invoke-Kerberoast at this abstraction layer might focus on the PowerShell function name (Invoke-Kerberoast), a string within the code (“@harmj0y”), or even the cryptographic hash of the script itself. These are great first steps and it would be silly to ignore these simple detections, but what happens when a new tool is released to accomplish the same attack technique?

Rubeus Kerberoast

Rubeus is a utility that includes many capabilities for abusing Kerberos, but the “kerberoast” command is specifically related to Kerberoasting. It turns out that Rubeus’s kerberoast command is functionally equivalent to Invoke-Kerberoast, but the capability is packaged a bit differently.

```
PS C:\Rubeus-master\Rubeus\obj\Debug> .\Rubeus.exe kerberoast /spn:mr3000/marvel.local /rc4opsec
```



```
v1.4.2
```

```
[*] Action: Kerberoasting
```

Rubeus kerberoast execution

The abstraction map can now be updated to include Rubeus’s kerberoast command to the tools layer.

While Invoke-Kerberoast is written in PowerShell, Rubeus is written in C#, so there may be differences in how detections can be implemented. For instance, the PowerShell function name and a cryptographic hash of Invoke-Kerberoast are not relevant to

detecting Rubeus. Instead, a detection engineer might focus on writing signatures for command-line arguments or maybe they would create additional detections using the cryptographic hash of Rubeus.

As you can imagine, this approach scales very poorly because of potentially infinite tool implementations variants for Kerberoasting. Instead, consider viewing tools as the most superficial layer of abstraction and dig into the actual functionality of Invoke-Kerberoast and Rubeus.

Managed Code

At this point in the journey, two unique Kerberoasting tools have been identified. Now it is important to think about what Invoke-Kerberoast and Rubeus kerberoast have in common. Those familiar with PowerShell and C# might remember that both languages are built on top of the .NET Framework, which is an abstraction itself that allows programmers to focus on functionality without worrying about complex programming topics like memory management, security, portability, etc. Keeping this in mind, it is reasonable to expect that these two tools might show some similarities when evaluated at the managed code layer. When comparing the code used to request service tickets from [Invoke-Kerberoast](#) and [Rubeus](#), it is revealed that they both rely on a call to a constructor overload for the [KerberosRequestorSecurityToken .NET Class](#) as shown below.

Invoke-Kerberoast calling the constructor for KerberosRequestorSecurityToken

Rubeus kerberoast calling the constructor for KerberosRequestorSecurityToken

As you can see, while the tools are written in different languages, they both eventually rely on the KerberosRequestorSecurityToken class. In theory, if a detection could be built to focus on the use of this particular class, a single detection could be used for both tool implementations.

Its time to add a new layer to the abstraction map for “Managed Code.” Notice that this layer has one entry that covers both tool implementations.

Hopefully, the value of peeling back the abstraction is beginning to become clear. However, it is important to note that while peeling back layers of abstractions allow for a more broad or comprehensive detection, it also means that the detection is possibly losing precision, which leads to a higher likelihood of false positives. For instance, what are the legitimate use cases for the KerberosRequestorSecurityToken class in .NET and how can analysts differentiate legitimate use from malicious use? This is a topic that a colleague of mine will discuss in more depth in a future blog post.

Windows API Functions

Imagine that you write detection logic to alert on the use of the `KerberosRequestorSecurityToken` .NET class. Should you stop there and consider Kerberoasting a solved problem? What happens if someone writes a Kerberoasting tool in a language that doesn't interact with .NET? Maybe they decide to write it in C++. Well, as mentioned earlier, .NET is an abstraction layer itself, so let's dig deeper to see what is happening underneath.

Luckily, .NET is open source, so the [source code](#) (.NET Reference Source Code) is readily accessible. Using the reference source, detection engineers can dig into the `KerberosRequestorSecurityToken` class's implementation to understand how it works under the hood. With a bit of investigation, it turns out that the `KerberosRequestorSecurityToken` class functionality (constructor) that is used by `InvokeKerberos` and `Rubeus's kerberoast` command is built on two Windows API functions from `Secur32.dll`, namely [AcquireCredentialsHandle](#) and [InitializeSecurityContext](#).

The Windows API is a set of functions that allow programmers to interact with the operating system in a predefined and documented way. Think of these as yet another layer of abstraction that dictates how software can interact with hardware components, such as memory, network interfaces, or input devices. It is also theoretically possible for attackers to call `InitializeSecurityContext` without flowing through the `KerberosRequestorSecurityToken` .NET class.

Below are screenshots from the .NET Reference Source showing how .NET implements [AcquireCredentialsHandleW](#) and [InitializeSecurityContextW](#) via a concept called [Platform Invoke \(P/Invoke\)](#).

.NET Reference Source — AcquireCredentialsHandleW

```
[DllImport(SEcurity, ExactSpelling = true, SetLastError = true)]
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.MayFail)]
[ResourceExposure(ResourceScope.None)]
internal extern static unsafe int InitializeSecurityContextW(
    ref SSPIHandle credentialHandle,
    [In] void* inContextPtr,
    [In] byte* targetName,
    [In] SspiContextFlags inFlags,
    [In] int reservedI,
    [In] Endianness endianness,
    [In] SecurityBufferDescriptor inputBuffer,
    [In] int reservedII,
    ref SSPIHandle outContextPtr,
    [In, Out] SecurityBufferDescriptor outputBuffer,
    [In, Out] ref SspiContextFlags attributes,
    out long timestamp
);
```

.NET Reference Source — InitializeSecurityContextW

As a result, this function provides yet another layer where detection engineers may be able to build detection logic.

T1208 - Kerberoasting		
Tool	PowerShell Invoke-Kerberoast	Rubeus kerberoast
Managed Code	.NET KerberosRequestorSecurityToken	
Windows API Function	InitializeSecurityContext	

A logical assumption might be that a Kerberoasting tool written in a language that doesn't use .NET would also use the InitializeSecurityContext API function. Let's take a look at a tool written in C++ and see if that assumption is correct.

Almost everyone in information security is familiar with [mimikatz](#) as a tool that allows attackers to dump plaintext passwords from memory. What fewer know is that mimikatz is a complex tool that provides numerous capabilities, one of which is the ability to request arbitrary service tickets for Kerberoasting. Below is an image that shows mimikatz being used to request the same service ticket that we requested with Invoke-Kerberoast and Rubeus.

```
mimikatz # kerberos::ask /target:mr3000/marvel.local
Asking for: mr3000/marvel.local
* Ticket Encryption Type & kvno not representative at screen

Start/End/MaxRenew: 1/18/2020 10:09:01 AM ; 1/18/2020 8:09:01 PM ; 1/25/2020 10:09:01 AM
Service Name (02) : mr3000 ; marvel.local ; @ MARVEL.LOCAL
Target Name (02) : mr3000 ; marvel.local ; @ MARVEL.LOCAL
Client Name (01) : thor ; @ MARVEL.LOCAL
Flags 40a10000 : name_canonicalize ; pre_authent ; renewable ; forwardable ;
Session Key : 0x00000017 - rc4_hmac_nt
e5a59cb763665dd725bb34d5989e5ab9
Ticket : 0x00000017 - rc4_hmac_nt ; kvno = 0 [...]
```

Mimikatz kerberos::ask command execution

Mimikatz's kerberos::ask command can now be added to the list of tools that can be used for Kerberoasting as shown below:

Now, how can a detection engineer determine if mimikatz builds on InitializeSecurityContext? Again this is a case where the tool is open source, so the source can be checked to see if mimikatz calls the InitializeSecurityContext.

Inspecting the source code for the kerberos::ask command reveals that it is implemented by the [kuhl_m_kerberos_ask](#) function.


```

const KUHL_M_C kuhl_m_c_kerberos[] = {
    {kuhl_m_kerberos_ptt,          L"ptt",          L"Pass-the-ticket [NT 6]"},
    {kuhl_m_kerberos_list,        L"list",         L"List ticket(s)"},
    {kuhl_m_kerberos_ask,         L"ask",          L"Ask or get TGS tickets"},
    {kuhl_m_kerberos_tgt,         L"tgt",          L"Retrieve current TGT"},
    {kuhl_m_kerberos_purge,       L"purge",        L"Purge ticket(s)"},
    {kuhl_m_kerberos_golden,      L"golden",       L"Willy Wonka factory"},
    {kuhl_m_kerberos_hash,        L"hash",         L"Hash password to keys"},
#ifdef KERBEROS_TOOLS
    {kuhl_m_kerberos_decode,       L"decrypt",      L"Decrypt encoded ticket"},
    {kuhl_m_kerberos_pac_info,     L"pacinfo",      L"Some infos on PAC file"},
#endif
    {kuhl_m_kerberos_ccache_ptc,   L"ptc",          L"Pass-the-ccache [NT6]"},
    {kuhl_m_kerberos_ccache_list, L"clist",        L"List tickets in MIT/Heimdall ccache"},
};

```

The kuhl_m_kerberos_ask function then calls a function called LsaCallKerberosPackage.

```

NTSTATUS kuhl_m_kerberos_ask(int argc, wchar_t * argv[])
{
    NTSTATUS status, packageStatus;
    PWCHAR filename = NULL, ticketname = NULL;
    PCWCHAR szTarget;
    PKERB_RETRIEVE_TKT_REQUEST pKerbRetrieveRequest;
    PKERB_RETRIEVE_TKT_RESPONSE pKerbRetrieveResponse;
    KIWI_KERBEROS_TICKET ticket = {0};
    DWORD szData;
    USHORT dwTarget;
    BOOL isExport = kull_m_string_args_byName(argc, argv, L"export", NULL, NULL), isTkt = kull_m_string_args_byName(argc, a

    if(kull_m_string_args_byName(argc, argv, L"target", &szTarget, NULL))
    {
        dwTarget = (USHORT) ((wcslen(szTarget) + 1) * sizeof(wchar_t));

        szData = sizeof(KERB_RETRIEVE_TKT_REQUEST) + dwTarget;
        if(pKerbRetrieveRequest = (PKERB_RETRIEVE_TKT_REQUEST) LocalAlloc(LPTR, szData))
        {
            pKerbRetrieveRequest->MessageType = KerbRetrieveEncodedTicketMessage;
            pKerbRetrieveRequest->CacheOptions = isNoCache ? KERB_RETRIEVE_TICKET_DONT_USE_CACHE : KERB_RETRIEVE_TI
            pKerbRetrieveRequest->EncryptionType = kull_m_string_args_byName(argc, argv, L"rc4", NULL, NULL) ? KERB
            pKerbRetrieveRequest->TargetName.Length = dwTarget - sizeof(wchar_t);
            pKerbRetrieveRequest->TargetName.MaximumLength = dwTarget;
            pKerbRetrieveRequest->TargetName.Buffer = (PWSTR) ((PBYTE) pKerbRetrieveRequest + sizeof(KERB_RETRIEVE_
            RtlCopyMemory(pKerbRetrieveRequest->TargetName.Buffer, szTarget, pKerbRetrieveRequest->TargetName.Maxim
            kprintf(L"Asking for: %wZ\n", &pKerbRetrieveRequest->TargetName);

            status = LsaCallKerberosPackage(pKerbRetrieveRequest, szData, (PVOID *) &pKerbRetrieveResponse, &szData

```

The LsaCallKerberosPackage function goes on to finally call a Windows API function called LsaCallAuthenticationPackage.

It appears that this logical assumption turned out to be incorrect. Instead of calling InitializeSecurityContext, mimikatz calls LsaCallAuthenticationPackage. It appears that there are at least two Windows API functions that can request service tickets. Let's take a moment and update our abstraction map.

By now you probably know the next question that needs to be answered! Is there common ground between InitializeSecurityContext and LsaCallAuthenticationPackage?

Remote Procedure Call

It appears that at least two Windows API functions that can be used to request a service ticket, but do those API functions converge to a shared abstraction layer underneath? To help figure this out, I worked with Matt Graeber ([@mattifestation](#)) to analyze security.dll (the library that both functions live in). During our analysis, it appeared that both InitializeSecurityContext and LsaCallAuthenticationPackage made RPC calls. Additional inspection showed that both API functions interact with the RPC interface with a UUID of 4f32adc8-6052-4a04-8701-293ccf2096f0.

A quick Google search of the UUID returned Matt Nelson's ([@enigma0x3](#)) [GitHub gist](#) where he listed all of the RPC interfaces on his machine. The screenshot below shows that the RPC interface in question is implemented by sspisrv.dll.

If you are interested in a deeper dive into how this was discovered, check out Adam Chester's ([@_xpn](#)) excellent post titled [Exploring Mimikatz Part 2](#). In his post, Adam explains how he examined a very similar situation involving the inner workings of mimikatz's Security Support Provider capability.

A new "RPC" layer can now be added to the abstraction map to show how both InitializeSecurityContext and LsaCallAuthenticationPackage ultimately rely on the same RPC call behind the scenes.

T1208 - Kerberoasting				
Tool	PowerShell Invoke-Kerberoast	Rubeus kerberoast	Mimikatz kerberos::ask	
Managed Code	.NET KerberosRequestorSecurityToken			
Windows API Function	InitializeSecurityContext		LsaCallAuthenticationPackage	
RPC	4f32adc8-6052-4a04-8701-293ccf2096f0 C:\WINDOWS\SYSTEM32\SspiSrv.dll			

Network Protocol

You might be asking yourself, "well wouldn't the attacker have to call one of the API functions that we talked about to make the network request in the first place?" The answer can be found in our old friend Rubeus. Rubeus has many bits of functionality that focus on abusing the Kerberos protocol. The "kerberoast" functionality was covered earlier, but a different command called "asktgs" allows an attacker to request a service

ticket by building a raw TGS-REQ packet and parsing the raw TGS-REP response. Think of this as the sustainable, local, handcrafted, artisanal approach to Kerberoasting. Rubeus asktgs is shown below:

```
PS C:\Tools\Rubeus\Rubeus\obj\Debug> .\Rubeus.exe asktgs /ticket:doIE8DCCBoygAwIBBaEDAgEwoOIEBjCCBAJhgP+MIID+qADAgEFoQ...
bDE1BULZFTC5MT0NBTKIhMB+gAwIBAAQEMBYbBmtyYnRndBsMbnWfydmVsLmxvY2Fso4IDVjCCA7qgAwIBEqEDAgECooIDrASCA6g17HKP0j6bG7mAh+IN2Q...
Djq5zln/Q9qnSat4rgJsX+Cxn4UHIbVvpV0nBG7e3hyZEjhESILQWUf0VT5yGwxr6MKSrg2Fht5MeV2bIAsEB3q+Ogg+16u5x2LyTY0398xPcyeus3/+kgY...
Ac4NPPoLx3r87ku89+sRkoP/lpIphWIEed1DiCDLaR0LFhUWDRXThZS+/1F8/DHbl3JNodtNhw5zcnuMHiXesnwr1SGkwhtyIffbaZLhfPdyfIyop3JEqwX...
t2nJVN7y2B11d8Qq85Q5KCHgWIquaPqzhyah6Pg20HNXojz7/jkHsATgpFCLHncIzvt4yJQ3rGXn1CJ9ZCJMjYnwnKBqONzYihAKt65T8PGDMjKqFYD317...
N8JTxrWlx+F0cxZLoaL5Vjm0r8S7LpxgqzNcXxRdDtbsE6Rmv5zsRH1HjZ3aQ/mBd3NchZDwARQX0kHfvXezB4QmYds1HcaNs0d/fy7b7Z1dLcf0QQR127W...
QstpgLbcwNG5GckfKp9n8k1mELWQjwbJt/gTIWgpbTj3KwTc3jkDq1LHmDmcwtnwvfaqrk1gMgeLV20fvw7DUU9F3B+2vk3nkXbjQPzjMuqsUznMZ6FF...
4t6fBGVrZtfr1GdTU6/Kv5W2aXwUofG7RbqMp6S1p/KEVAfGaycOT8gEh1Hw3LNFLy25WC3meq+H2eLrkUnBie6hg/qLczw4+SSNFgZErU3gIi13ESR5qb...
8apRe8rkuL1mbVf2j569gSpqNDU+ytB7JrKMj7z1yc/O+e2TQMZ14FrIN31CRHrrDPtm8M+fhsWIXwAsk0M+/vZBqY0/RN3U+kN7iU8ibeF1foj0a25dglc...
d9vBnaF81uDwWxDQ2xvdgtMujkPFV5IpsrW2+DEzd1ub1wt04WtP3vicdex79dYiObGw6h1F980i8ufnJBaNC9H1YhJwoz8Ebi82JbWy/nR/JS/3DX0u91...
8JcNwRYjgC0dotu/H876I1XLPVfBwP/gqyb31sc1bndC1vqiQTBb69rGdbu7iVEjNnZqxBv11i0C7NEN4Ic/qXmURCBVrd4hS91cvtRMMjEz4EattD067/k...
oFRq43kSW0ZVPnSEfnjUHoVE8zy02/LpA2xGdrT5c3d/r137waY010wBBVcyLJwBKgtUZU12upIXgT2e2GgS6rLxUwL7EcaccXCXxIBWwKcYZ8YIr9vY3...
zeumpWrtDjIydmeEZur2izkro4jGjgdUwgdKgAwIBAKKBygSBx32BxDCBwaCBvjCBuzCBuKAbMBmgAwIBF6ESBBDIczZUU8V2bGV4WTYa1R65oQ4bDE1BU1...
FTCSMT0NBTKIUMBKAwIBAAELMAkb21yb25tYw6jBwMFAEDhAAC1ERgPMjAyMDAxMTgxNzMSMDVaphEYDzIwMjAwMTE5MDMzOTA1WqcRGA8yMDIwMDEyNT...
3MzkNwVqoDhsMTUFSVkvMLkxPQ0FMqSEW6ADAgECoRgwFhsGa3JidGd0GwxtYXJ2ZWwubG9jYWw= /service:mr3000/marvel.local enctype:rc4
```

Rubeus asktgs execution

The abstraction map must be updated to include Rubeus's asktgs at the tool layer. Unfortunately, this implementation has no overlaps with any of the abstraction layers the have been discovered thus far. This sounds like an opportunity for more digging!

Fundamentally, Kerberos is a network authentication protocol where a user authenticates with a central entity, called the Kerberos Key Distribution Center (KDC). Kerberoasting requires that a service ticket or multiple service tickets be requested to facilitate offline password cracking. To make this request, a network connection from the client to the KDC must be made and the network request must be in the form of a Ticket Granting Service Request (TGS-REQ).

Below, are packet captures of Invoke-Kerberoast, Rubeus kerberoast, mimikatz kerberos::ask, and Rubeus asktgs. Notice that while there are slight differences in the packet captures, each tool makes the same TGS-REQ request, which is highlighted in each image.

Wireshark capture of Invoke-Kerberoast

Wireshark capture of Rubeus kerberoast

tcp_stream eq 4

No.	Time	Source	Destination	Protocol	Length	Info
33	2020-01-18 18:09:01.859815	192.168.38.102	192.168.38.101	TCP	66	50079 → 88 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
35	2020-01-18 18:09:01.859980	192.168.38.101	192.168.38.102	TCP	66	88 → 50079 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
36	2020-01-18 18:09:01.860018	192.168.38.102	192.168.38.101	TCP	54	50079 → 88 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
37	2020-01-18 18:09:01.860046	192.168.38.102	192.168.38.101	KRB5	1612	TGS-REQ
38	2020-01-18 18:09:01.860238	192.168.38.101	192.168.38.102	TCP	60	88 → 50079 [ACK] Seq=1 Ack=1559 Win=525568 Len=0
39	2020-01-18 18:09:01.860741	192.168.38.101	192.168.38.102	TCP	1514	88 → 50079 [ACK] Seq=1 Ack=1559 Win=525568 Len=1460 [TCP segment of a reassembled
40	2020-01-18 18:09:01.860742	192.168.38.101	192.168.38.102	KRB5	129	TGS-REP
41	2020-01-18 18:09:01.860760	192.168.38.102	192.168.38.101	TCP	54	50079 → 88 [ACK] Seq=1559 Ack=1536 Win=2102272 Len=0
42	2020-01-18 18:09:01.860844	192.168.38.102	192.168.38.101	TCP	54	50079 → 88 [FIN, ACK] Seq=1559 Ack=1536 Win=2102272 Len=0
43	2020-01-18 18:09:01.861128	192.168.38.101	192.168.38.102	TCP	60	88 → 50079 [ACK] Seq=1536 Ack=1560 Win=525568 Len=0
44	2020-01-18 18:09:01.861129	192.168.38.101	192.168.38.102	TCP	60	88 → 50079 [RST, ACK] Seq=1536 Ack=1560 Win=0 Len=0

Frame 37: 1612 bytes on wire (12896 bits), 1612 bytes captured (12896 bits) on interface 0
 Ethernet II, Src: Vmware_b8:29:10 (00:0c:29:bb:29:10), Dst: Vmware_51:6c:e4 (00:0c:29:51:6c:e4)
 Internet Protocol Version 4, Src: 192.168.38.102, Dst: 192.168.38.101
 Transmission Control Protocol, Src Port: 50079, Dst Port: 88, Seq: 1, Ack: 1, Len: 1558
 Kerberos
 Record Mark: 1554 bytes
 tgs-req
 pvno: 5
 msg-type: krb-tgs-req (12)
 padata: 2 items
 req-body
 Padding: 0
 kdc-options: 40810000 (forwardable, renewable, canonicalize)
 realm: MARVEL.LOCAL
 sname
 name-type: kRB5-NT-SRV-INST (2)
 sname-string: 2 items
 SNameString: mr3000
 SNameString: marvel.local
 till: 2037-09-13 02:48:05 (UTC)
 nonce: 1565011684
 etype: 5 items

SEQUENCE_OF_SNameString (kerberos.sname_string), 22 bytes

Wireshark capture of Mimikatz kerberos:ask

tcp_stream eq 2

No.	Time	Source	Destination	Protocol	Length	Info
21	2020-01-18 18:01:06.641092	192.168.38.102	192.168.38.101	TCP	66	50049 → 88 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
22	2020-01-18 18:01:06.641451	192.168.38.101	192.168.38.102	TCP	66	88 → 50049 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
23	2020-01-18 18:01:06.641533	192.168.38.102	192.168.38.101	TCP	54	50049 → 88 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
24	2020-01-18 18:01:06.641628	192.168.38.102	192.168.38.101	KRB5	1395	TGS-REQ
25	2020-01-18 18:01:06.645544	192.168.38.101	192.168.38.102	KRB5	1398	TGS-REP
26	2020-01-18 18:01:06.645723	192.168.38.102	192.168.38.101	TCP	54	50049 → 88 [FIN, ACK] Seq=1342 Ack=1345 Win=2100992 Len=0
27	2020-01-18 18:01:06.646002	192.168.38.101	192.168.38.102	TCP	60	88 → 50049 [ACK] Seq=1345 Ack=1343 Win=525568 Len=0
28	2020-01-18 18:01:06.646939	192.168.38.101	192.168.38.102	TCP	60	88 → 50049 [RST, ACK] Seq=1345 Ack=1343 Win=0 Len=0

Frame 24: 1395 bytes on wire (11160 bits), 1395 bytes captured (11160 bits) on interface 0
 Ethernet II, Src: Vmware_b8:29:10 (00:0c:29:bb:29:10), Dst: Vmware_51:6c:e4 (00:0c:29:51:6c:e4)
 Internet Protocol Version 4, Src: 192.168.38.102, Dst: 192.168.38.101
 Transmission Control Protocol, Src Port: 50049, Dst Port: 88, Seq: 1, Ack: 1, Len: 1341
 Kerberos
 Record Mark: 1337 bytes
 tgs-req
 pvno: 5
 msg-type: krb-tgs-req (12)
 padata: 1 item
 req-body
 Padding: 0
 kdc-options: 40800010 (forwardable, renewable, renewable-ok)
 cname
 realm: MARVEL.LOCAL
 sname
 name-type: kRB5-NT-SRV-INST (2)
 sname-string: 2 items
 SNameString: mr3000
 SNameString: marvel.local
 till: 2037-09-12 19:48:05 (UTC)
 nonce: 1818848256

SEQUENCE_OF_SNameString (kerberos.sname_string), 22 bytes

Wireshark capture of Rubeus asktgs

The abstraction map is completed by adding a Network Protocol layer that covers all implementations of Kerberoasting. Keep in mind that while the RPC and Network Protocol layers are more inclusive of Kerberoasting implementations, they are also more inclusive of legitimate activity. Simply put, it is not always ideal to build detections at lower abstraction layers.

T1208 - Kerberoasting				
Tool	PowerShell Invoke-Kerberoast	Rubeus kerberoast	Mimikatz kerberos::ask	Rubeus asktgs
Managed Code	.NET KerberosRequestorSecurityToken			
Windows API Function	InitializeSecurityContext			
RPC	4f32adc8-6052-4a04-8701-293ccf2096f0 C:\WINDOWS\SYSTEM32\SspiSrv.dll			
Network Protocol	Kerberos TGS-REQ/REP			

Keep in mind that this abstraction map may not consider every possible abstraction involved in Kerberoasting. Can you think of any additional abstraction layers? If so, please share them in the comments!

Conclusion

I've commonly heard attackers talk about how important it is to understand how their attack tools work, but I've rarely heard a similar sentiment from my defensive family. Too often I see a detection engineer take an attack tool at face value. Understanding how offensive capabilities are abstracted creates an opportunity for detection engineers to evaluate their detection approach in an informed way. I believe that by taking more time to understand the technology that tools abstract away from us, we will be able to raise the bar and not just detect what we know about, but what we don't know about as well. Capability Abstraction is just one tool that you can add to your detection engineering tool chest.

The next posts in this series will explain strategies for understanding the pros and cons of detections at different layers of abstraction and how to leverage an abstraction map to determine the most effective approach to layering detection logic to create a comprehensive detection approach.