September 26, 2016

Rotten Potato – Privilege Escalation from Service Accounts to SYSTEM

By @breenmachine

This past Friday, myself and my partner in crime, Chris Mallz (@vvalien1) spoke at DerbyCon about a project we've been working on for the last few months. For those interested in watching the talk, it's online here and the code is available on the FoxGlove Security GitHub page.

This blog post is going to dive into some of the technical details of our project in order to remediate @singe's very accurate observation about our README file on Twitter:

In reply to Steve Breen
Dominic White @singe · Sep 24
@breenmachine @TheColonial @vvalien1 @DerbyCon nothing says "late night, last minute, new features" like that Readme ;)
2

So without further delay…

Overview

As we mentioned a number of times throughout our talk, this work is derived directly from James Forshaw's BlackHat talk and Google Project Zero research. I highly recommend reviewing both of these resources to anyone interested in pursuing this topic.

The idea behind this vulnerability is simple to describe at a high level:

1. Trick the "NT AUTHORITY\SYSTEM" account into authenticating via NTLM to a TCP endpoint we control.
2. Man-in-the-middle this authentication attempt (NTLM relay) to locally negotiate a security token for the "NT AUTHORITY\SYSTEM" account. This is done through a series of Windows API calls.
3. Impersonate the token we have just negotiated. This can only be done if the attackers current account has the privilege to impersonate security tokens. This is usually true of most service accounts and not true of most user-level accounts.

Each of these steps are described in the following 3 sections.

NTLM Relay to Local Negotiation

NTLM relay from the local "NT AUTHORITY\SYSTEM" (we will just call it SYSTEM for brevity) account back to some other system service has been the theme for the Potato privilege escalation exploits. The first step is to trick the SYSTEM account into performing authentication to some TCP listener we control.

In the original Hot Potato exploit, we did some complex magic with NBNS spoofing, WPAD, and Windows Update services to trick it into authenticating to us over HTTP. For more information, see the original blog post.

Today, we'll be discussing another method to accomplish the same end goal which James Forshaw discussed here. We'll basically be tricking DCOM/RPC into NTLM authenticating to us. The advantage of this more complex method is that it is 100% reliable, consistent across Windows versions, and fires instantly rather than sometimes having to wait for Windows Update.

Getting Started

We'll be abusing an API call to COM to get this all kicked off. The call is "CoGetInstanceFromIStorage" and to give you some context, here is the relevant code:

```
public static void BootstrapComMarshal()
{
IStorage stg = ComUtils.CreateStorage();
// Use a known local system service COM server, in this cast
BITSv1
Guid clsid = new Guid("4991d34b-80a1-4291-83b6-3328366b9097");

TestClass c = new TestClass(stg, String.Format("{0}[{1}]",
"127.0.0.1", 6666)); // ip and port

MULTI_QI[] qis = new MULTI_QI[1];

qis[0].pIID = ComUtils.IID_IUnknownPtr;
qis[0].pItf = null;
qis[0].hr = 0;

CoGetInstanceFromIStorage(null, ref clsid, null,
CLSCTX.CLSCTX_LOCAL_SERVER, c, 1,       qis);
}
```

I’m far from being an expert on COM. The “CoGetInstanceFromIStorage” call attempts to fetch an instance of the specified object from a location specified by the caller. Here we are telling COM we want an instance of the BITS object and we want to load it from 127.0.0.1 on port 6666.

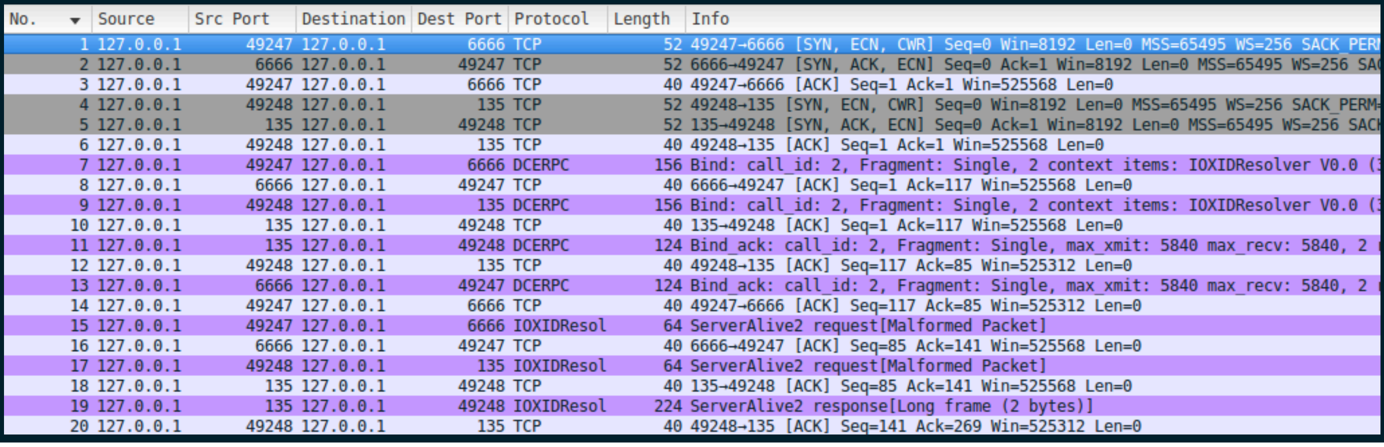
It’s actually a little more complex than that, because really we’re fetching the object from an “IStorage” object, not just passing a host/port directly. In the code above “TestClass” is actually an instance of an IStorage object in which we’ve replaced some bits and pieces to point back to “127.0.0.1:6666”.

Man-In-The-Middle

So, now we have COM trying to talk to us on port 6666 where we've spun up a local TCP listener. If we reply in the correct way, we have have COM (running as the SYSTEM account) try to perform NTLM authentication with us.

COM is trying to talk to us using the RPC protocol. I'm not particularly fluent in RPC and wouldn't be surprised if there were slight variations based on Windows versions. In order to avoid many headaches, we're going to use a trick in order to craft our replies. What we will do is relay any packets we receive from COM on TCP port 6666, back to the local Windows RPC listener on TCP port 135. Since these packets we're receiving are part of a valid RPC conversation, whatever version of Windows we are running will respond appropriately. We can then use these packets we receive back from Windows RPC on TCP 135 as templates for our replies to COM.

If that's not clear, the following shows the first few packets of this exchange in WireShark:



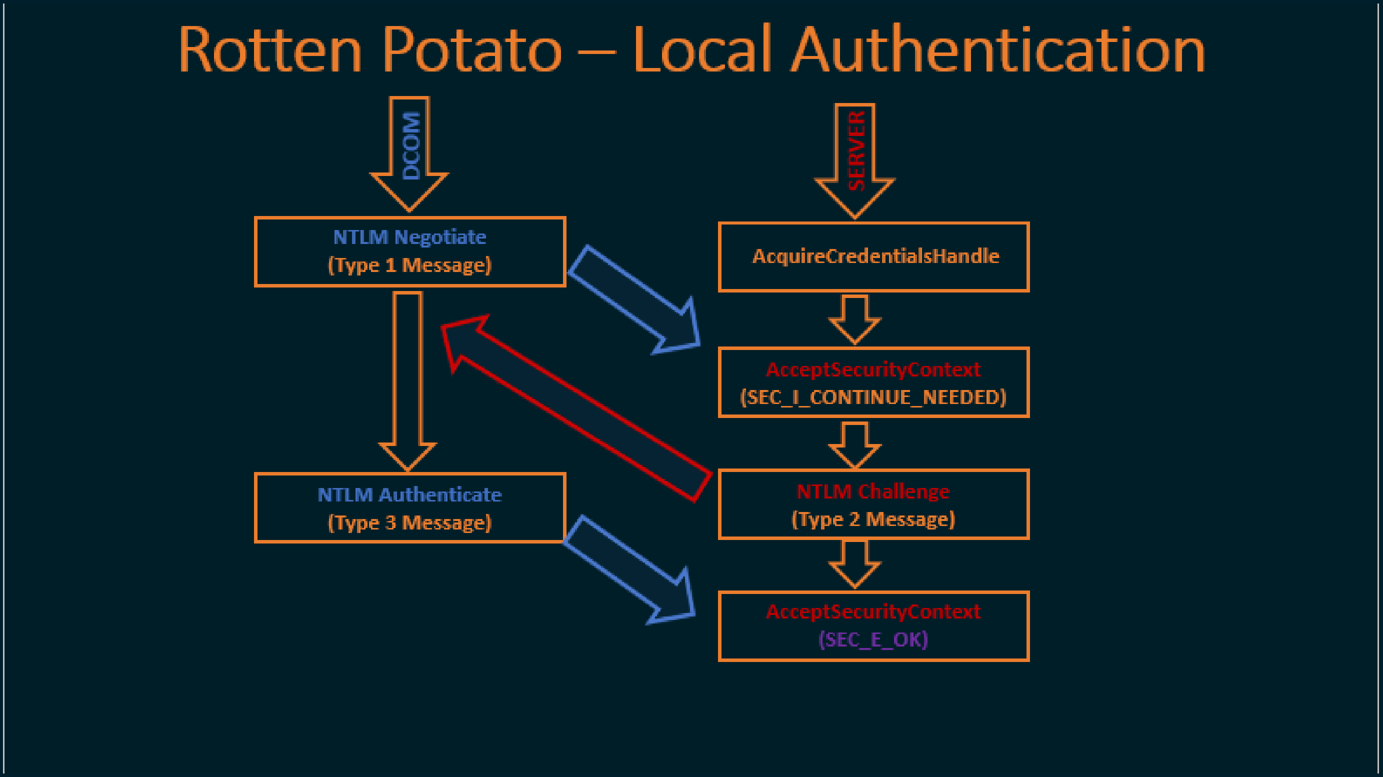
No.	Source	Src Port	Destination	Dest Port	Protocol	Length	Info
1	127.0.0.1	49247	127.0.0.1	6666	TCP	52	49247→6666 [SYN, ECN, CWR] Seq=0 Win=8192 Len=0 MSS=65495 WS=256 SACK_PER
2	127.0.0.1	6666	127.0.0.1	49247	TCP	52	6666→49247 [SYN, ACK, ECN] Seq=0 Ack=1 Win=8192 Len=0 MSS=65495 WS=256 SA
3	127.0.0.1	49247	127.0.0.1	6666	TCP	40	49247→6666 [ACK] Seq=1 Ack=1 Win=525568 Len=0
4	127.0.0.1	49248	127.0.0.1	135	TCP	52	49248→135 [SYN, ECN, CWR] Seq=0 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM
5	127.0.0.1	135	127.0.0.1	49248	TCP	52	135→49248 [SYN, ACK, ECN] Seq=0 Ack=1 Win=8192 Len=0 MSS=65495 WS=256 SAC
6	127.0.0.1	49248	127.0.0.1	135	TCP	40	49248→135 [ACK] Seq=1 Ack=1 Win=525568 Len=0
7	127.0.0.1	49247	127.0.0.1	6666	DCERPC	156	Bind: call_id: 2, Fragment: Single, 2 context items: IOXIDResolver V0.0 (
8	127.0.0.1	6666	127.0.0.1	49247	TCP	40	6666→49247 [ACK] Seq=1 Ack=117 Win=525568 Len=0
9	127.0.0.1	49248	127.0.0.1	135	DCERPC	156	Bind: call_id: 2, Fragment: Single, 2 context items: IOXIDResolver V0.0 (
10	127.0.0.1	135	127.0.0.1	49248	TCP	40	135→49248 [ACK] Seq=1 Ack=117 Win=525568 Len=0
11	127.0.0.1	135	127.0.0.1	49248	DCERPC	124	Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 5840, 2
12	127.0.0.1	49248	127.0.0.1	135	TCP	40	49248→135 [ACK] Seq=117 Ack=85 Win=525312 Len=0
13	127.0.0.1	6666	127.0.0.1	49247	DCERPC	124	Bind_ack: call_id: 2, Fragment: Single, max_xmit: 5840 max_recv: 5840, 2
14	127.0.0.1	49247	127.0.0.1	6666	TCP	40	49247→6666 [ACK] Seq=117 Ack=85 Win=525312 Len=0
15	127.0.0.1	49247	127.0.0.1	6666	IOXIDResol	64	ServerAlive2 request[Malformed Packet]
16	127.0.0.1	6666	127.0.0.1	49247	TCP	40	6666→49247 [ACK] Seq=85 Ack=141 Win=525568 Len=0
17	127.0.0.1	49248	127.0.0.1	135	IOXIDResol	64	ServerAlive2 request[Malformed Packet]
18	127.0.0.1	135	127.0.0.1	49248	TCP	40	135→49248 [ACK] Seq=85 Ack=141 Win=525568 Len=0
19	127.0.0.1	135	127.0.0.1	49248	IOXIDResol	224	ServerAlive2 response[Long frame (2 bytes)]
20	127.0.0.1	49248	127.0.0.1	135	TCP	40	49248→135 [ACK] Seq=141 Ack=269 Win=525312 Len=0

Notice that the first packet we receive (packet #7) is incoming on port 6666 (our listener, this is COM talking to us). Next, we relay that same packet (packet #9) to RPC on TCP 135. Then in packet #11, we get a reply back from RPC (TCP 135), and in packet #13, we relay that reply to COM.

We simply repeat this process until it's time for NTLM authentication to occur. You can think of these initial packets as just setting the stage for the eventual NTLM auth.

NTLM Relay and Local Token Negotiation

Before we dive into the NTLM relay details, let's look at it at a high level. The following is from our slide deck:


Rotten Potato – Local Authentication
DCOM
SERVER
NTLM Negotiate (Type 1 Message)
AcquireCredentialsHandle
AcceptSecurityContext (SEC_I_CONTINUE_NEEDED)
NTLM Challenge (Type 2 Message)
NTLM Authenticate (Type 3 Message)
AcceptSecurityContext (SEC_E_OK)

On the left in blue are the packets that COM is going to send to us on TCP port 6666. On the right, in red, are the Windows API calls that we're going to make using data that we pull out of those packets.

Let's look a little closer at the API calls on the right, since most people will not be familiar with them. In order to locally negotiate a security token using NTLM authentication, one must first call the function "AcquireCredentialsHandle" to get a handle to the data structure we will need.

Next, we call "AcceptSecurityContext", and the input to this function will be the NTLM Type 1 (Negotiate) message. The output will be an NTLM Type 2 (Challenge) message which is sent back to the client trying to authenticate, in this case, DCOM.

When the client responds with an NTLM Type 3 (Authenticate) message, we then pass that to a second call to "AcceptSecurityContext" to complete the authentication process and get a token.

Let's look at the packet capture and break this all down…

TYPE 1 (NEGOTIATE) PACKET

After relaying a few packets between RPC and COM, eventually COM is going to try to initiate NTLM authentication with us by sending the NTLM Type 1 (Negotiate) message, as shown in packet #29 of the packet capture below:

This is where things start to get interesting. Again, we relay this to RPC (on TCP 135), and RPC will reply with an NTLM Challenge.

But there's one more thing going on here that you don't see in the packet capture. When we receive the NTLM Type 1 (Negotiate) message from COM, we rip out the NTLM section of the packet (as shown below), and use it to begin the process of locally negotiating a token:

So, as discussed above, we call "AcquireCredentialsHandle", and then "AcceptSecurityContext", passing as input the NTLM Type 1 (Negotiate) message we pulled out of that packet.

NTLM TYPE 2 (CHALLENGE) PACKET

Recall that we forwarded the NTLM Type 1 (Negotiate) packet to RPC on port 135, RPC will now reply with an NTM Type 2 (Challenge) packet which can be seen in our packet capture above in packet #33. This time, we do NOT simply forward this packet back to COM, we need to do some work first.

Let's take a closer look at the two NTLM Type 2 (Challenge) packets from the capture above:

Notice the highlighted field "NTLM Server Challenge" and the field below it "Reserved", and that they differ in value. This would not be the case if we had simply forwarded the packet from RPC (on the left) to COM (the one on the right).

Recall that when we made the Windows API call to "AcceptSecurityContext", the output of that call was an NTLM Type 2 (Challenge) message. What we've done here is replace the NTLM blob inside the packet that we are sending to COM with the result of that API call.

Why would we do this? Because we need COM, running as the SYSTEM account to authenticate using the NTLM challenge and "Reserved" section that we are using to negotiate our local token, if we did not replace this section in the packet, then our call to "AcceptSecurityContext" would fail.

We'll talk more about how local NTLM authentication works later, but for now just know that the client who is trying to authenticate (in this case SYSTEM through COM) needs to do some magic with the "NTLM Server Challenge" and "Reserved" sections of the NTLM Type 2 (Negotiate) packet, and that we'll only get our token if this magic is performed on the values produced by our call to "AcceptSecurityContext".

NTLM TYPE 3 (AUTHENTICATE) PACKET

So now we've forwarded the modified NTLM Type 2(Negotiate) packet to COM where the "Challenge" and "Reserved" fields match the output from "AcceptSecurityContext". The "Reserved" field is actually a reference to a SecHandle, and when the SYSTEM account receives the NTLM Type 2 message, it will perform authentication behind the scenes in memory. That is why it is so crucial that we update the "Reserved" field… Otherwise, it would be authenticating to RPC instead of US!

Once this is completed, COM on behalf of the SYSTEM account will send us back the NTLM Type 3 (Authenticate) packet. This will just be empty (because all the actual authentication here happened in memory), but we will use it to make our final call to "AcceptSecurityContext".

We can then call "ImpersonateSecurityContext" with the result of the final call above to get an impersonation token.

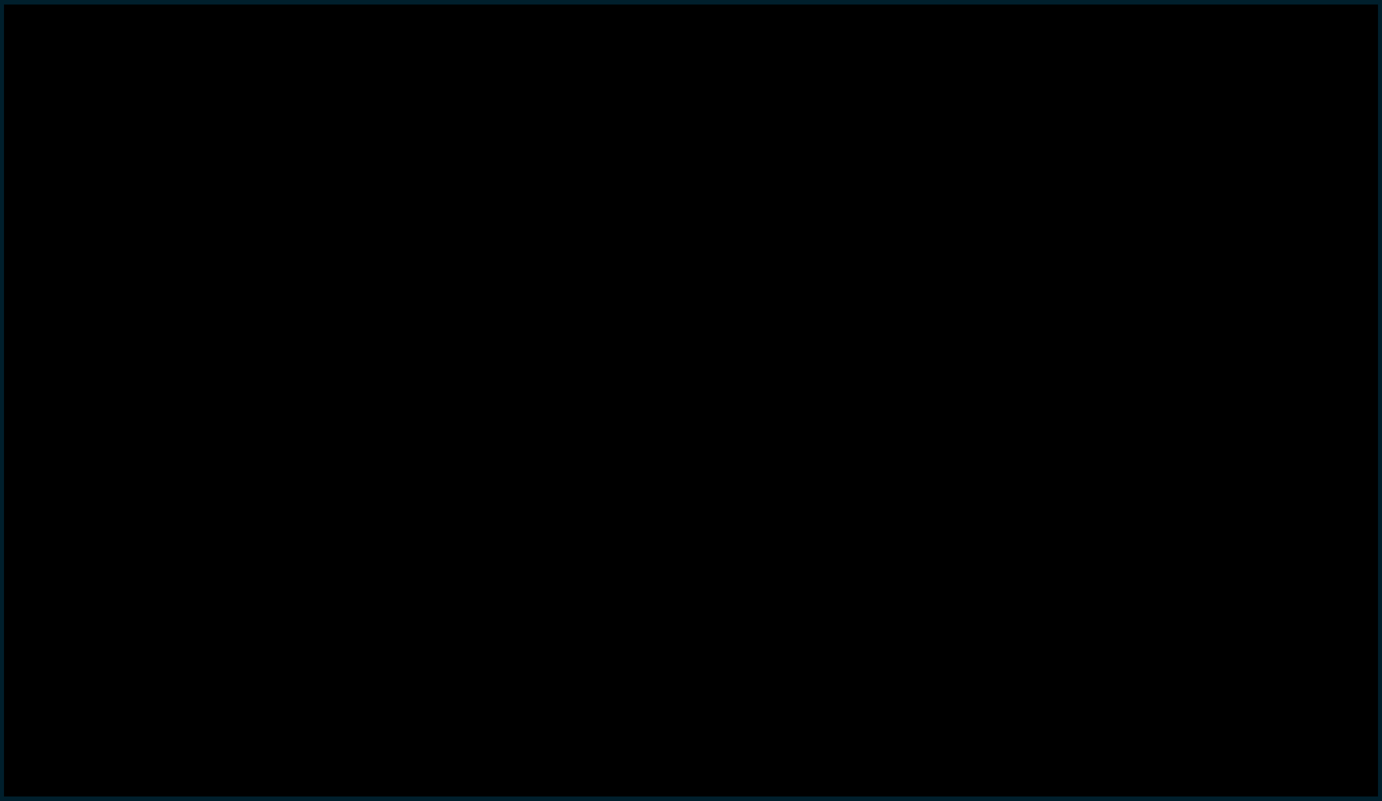
Using the ImpersonationToken

The following diagram (youtube play bar included) from James Forshaw's BlackHat talk "Social Engineering the Windows Kernel" shows the pre-requisites to impersonating the token that we have now negotiated:

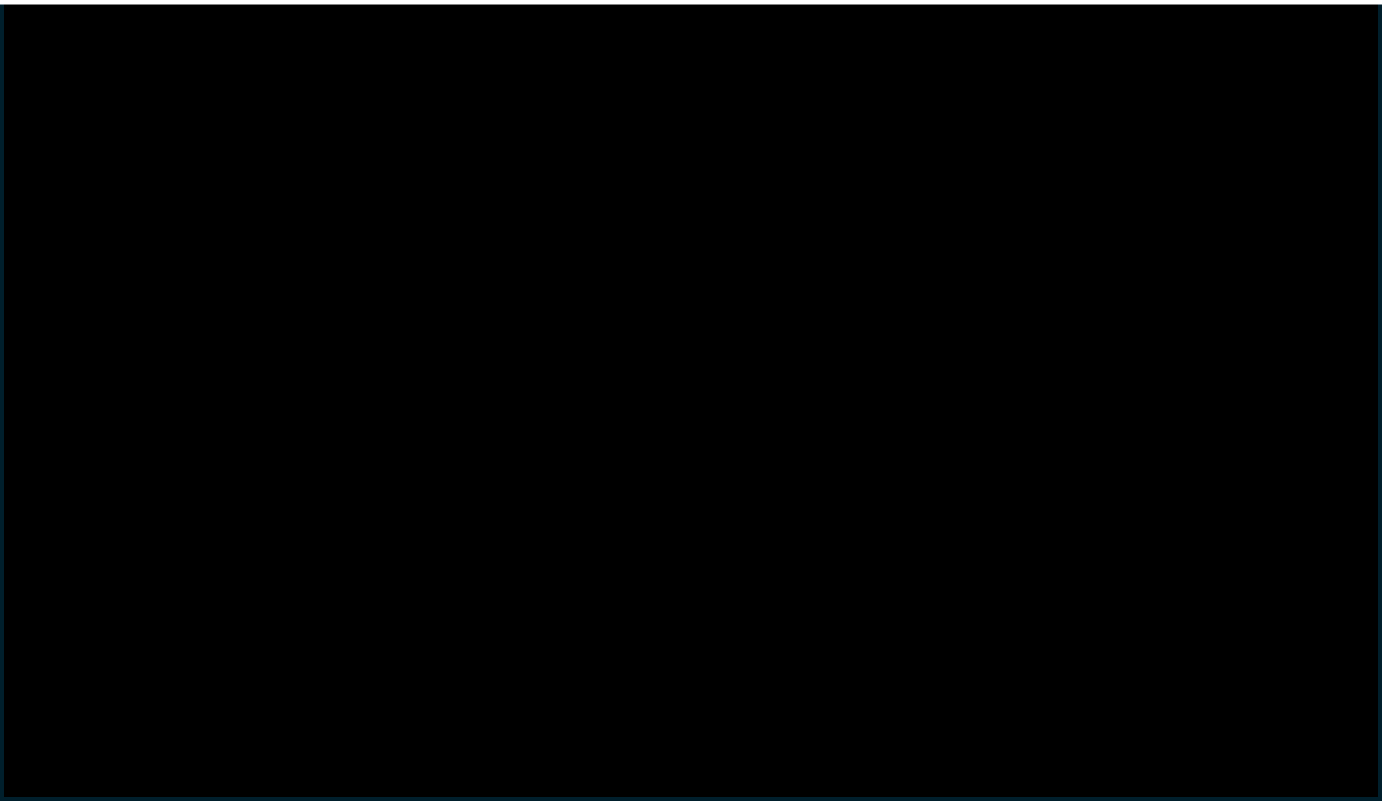
From this, it is clear that if we want to impersonate the token, we better be running as an account with SeImpersonate privilege (or equivalent). Luckily this includes many service accounts in Windows that penetration testers often end up running as. For example, the IIS and SQL Server accounts.

The following two videos show the exploit in action:

IIS:



SQL Server:



Share this:

Twitter Facebook

Loading...

arls; fuzzing ClamAV

sperReports – The Hidden Shell Feature