Alex Ionescu's Blog

Windows Internals, Thoughts on Security, and Reverse Engineering

---

« How Control Flow Guard Drastically Caused Windows 8.1 Address Space and Behavior Changes
What are Little PatchGuards Made Of? »

Analyzing MS15-050 With Diaphora

One of the most common ways that I glean information on new and upcoming features on releases of Windows is obviously to use reverse engineering such as IDA Pro and look at changed functions and variables, which usually imply a change in functionality.

Of course, such changes can also reveal security fixes, but those are a lot harder to notice at the granular level of diff-analysis that I perform as part of understanding feature changes.

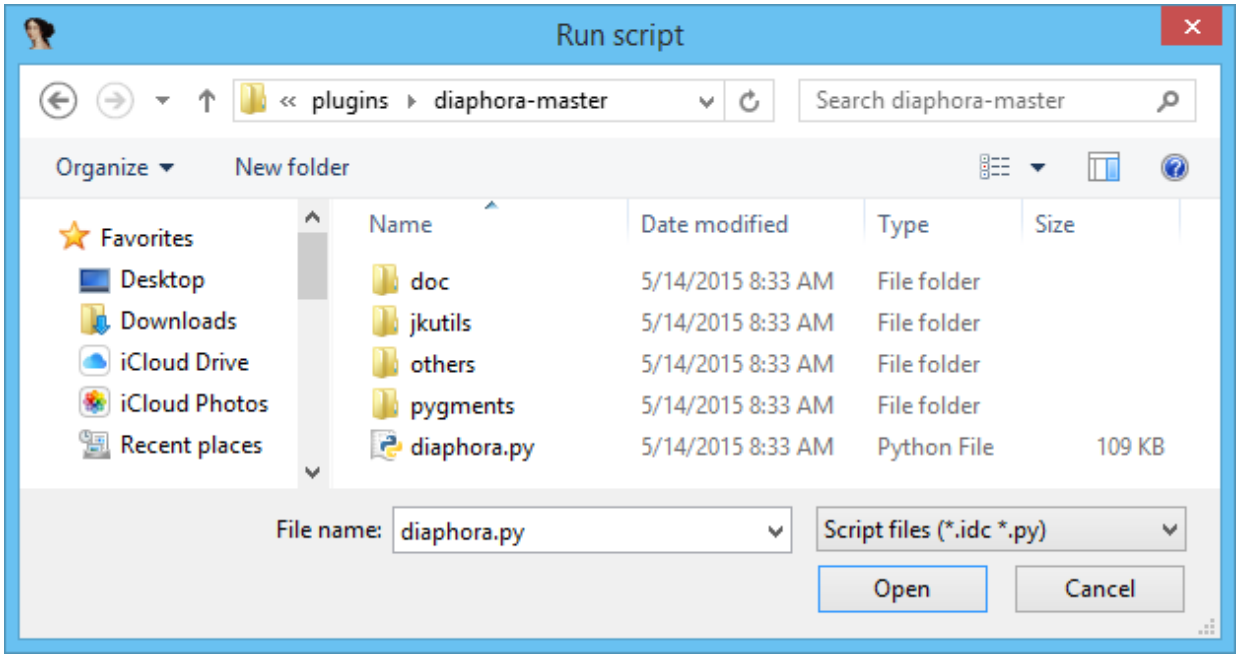
For those type of fixes, a specialized diffing tool, such as BinDiff is often used by reverse engineers and security experts. Recently, such tools have either become obsoleted, abandoned, or cost prohibitive. A good friend of mine, Joxean Koret (previously of Hex-Rays fame, un-coincidentally), has recently developed a Python plugin for IDA Pro, called "Diaphora", (diaforá, the Greek word for "difference").

In this blog post, we'll analyze the recent MS15-050 patch and do a very quick walk-through of how to use Diaphora.

Installation

Installing the plugin is as easy as going over to the GitHub page, cloning the repository into a .zip file, and extracting the contents into the appropriate directory (I chose IDA's plugin folder, but this can be anything you wish).

As long as your IDA Python is configured correctly (which has been a default in IDA for many releases), clicking on File, Script file…, should let you select a .py file



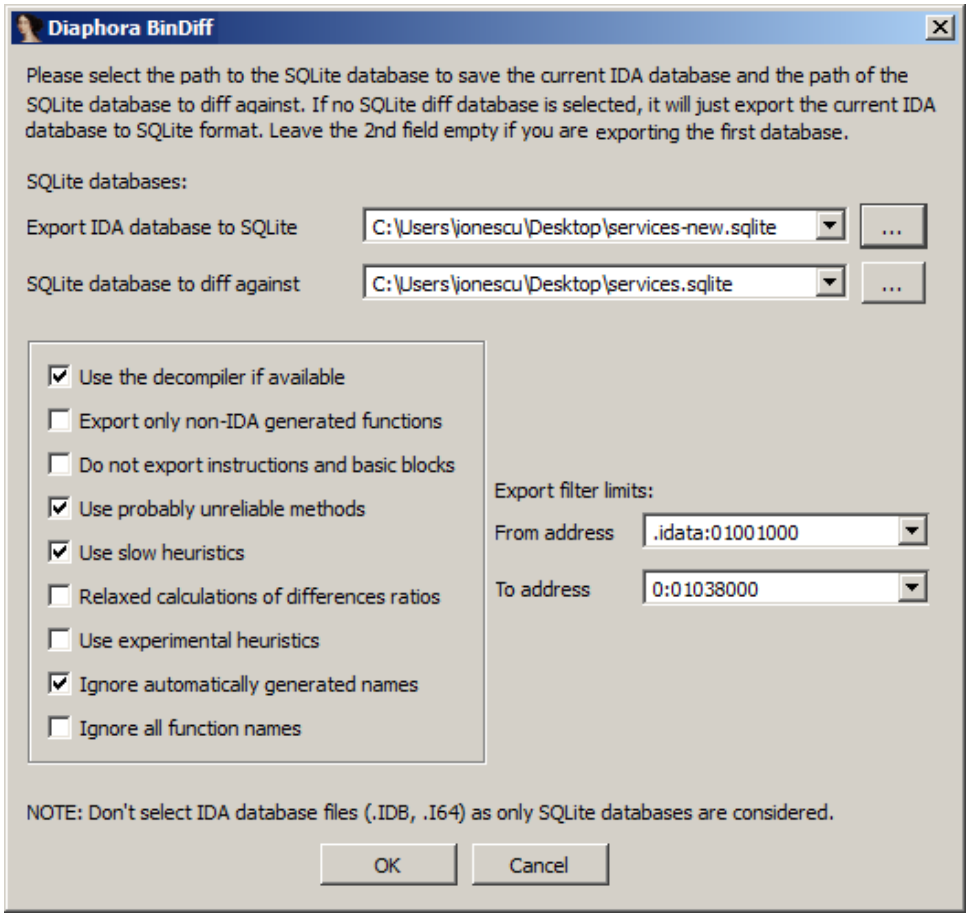
Generating the initial baseline

The first time you run Diaphora, you'll be making the initial SQLite library. If you don't have Hex-Rays, or disable the "Use the decompiler if available" flag, this process only takes a few seconds. Otherwise, with Hex-Rays enabled, you'll be spending more of the time waiting for the decompiler to run on the entire project. Depending on code complexity, this could take a while.

This SQLite library will essentially contain the assembly and pseudo-code in a format easily parsable by the plugin, as well as all your types, enumerations, and decompiler data (such as your annotations and renamed variables). In this case, I had an existing fairly well-maintained IDB for the latest version of the Service Control Manager for Windows 7 SP1, which had actually not changed since 2012. My pseudo-code had over 3 years to grow into a well-documented, thoroughly structured IDA database.

Diff me once, importing your metadata

On the second run of Diaphora (which at this point, should be on your new, fresh binary), this is where you will direct it to the initial SQLite database from the step above, plus select your diffing options. The default set I use are in the screenshot below.



This second run can take much longer than the first, because not only are you taking the time to generate the a second database, but you are then running all of the diffing algorithms that Diaphora implements (which you can customize), which can take significantly longer. Once the run is complete, Diaphora will show you identical code ("Best Matches"), close matches ("Partial Matches"), and Unidentifiable Matches. This is where comparing a very annotated IDB with a fresh IDB for purposes of security research can have problems.

Since I renamed many of the static global variables, any code using them in their renamed format would appear different from the original "loc_325345″ format that IDA uses by default. Any function prototypes which I manually fixed up would also appear different (Hex-Rays is especially bad with variable argument __stdcall on x86), as well any callers of those functions.

So in the initial analysis, I got tons of "Partial Matches" and very few "Best Matches". Nothing was unmatched, however.

One of the great parts of Diaphora, however, is that you can then confirm that the functions are truly identical. Since we're talking about files which have symbols, it makes sense to claim that *ScmFooBar* is identical to *ScmFooBar*. This will now import all the metadata from your first first IDB to the other, and then give you the option of re-running the analysis stage.

At this point, I have taken all of the 3 years of research I had on one IDB, and instantly (well, almost) merged it to a brand new IDB that covers a more recent version of the binary.

Diff me twice, locating truly changed code

Now that the IDBs have been "synced up", the second run should identify true code changes — new variables that have been added, structures that changed, and new code paths. In truth, those were

identified the first time around, but hidden in the noise of all the IDB annotation changes. Here's an incredible screenshot of what happened the second time I ran Diaphora.

First, note how almost all the functions are now seen as identical:

IDA View-A | Pseudocode-A | Diff pseudo-code ScStatusAccessCheck - ScStatusAccessCheck | Best matches | Partial matches

Line	Address	Name	Address 2	Name 2	Ratio	Description
00000	0100275d	??0CServiceRecordExclusiveLock@...	0100275d	??0CServiceRecordExclusiveLock@...	1.000	100% equal
00001	010027a1	?ScIsValidScManagerOrServiceHand...	010027a1	?ScIsValidScManagerOrServiceHand...	1.000	100% equal
00002	010028bb	?ScIsValidScManagerHandle@@YGH...	010028bb	?ScIsValidScManagerHandle@@YGH...	1.000	100% equal
00003	010028fa	_NdrServerCall2@4	010028fa	_NdrServerCall2@4	1.000	100% equal
00004	01002905	_memset	01002905	_memset	1.000	100% equal
00005	010029b6	??0CServiceRecordSharedLock@@...	010029b6	??0CServiceRecordSharedLock@@...	1.000	100% equal
00006	010029d1	??0CServiceListSharedLock@@QAE...	010029d1	??0CServiceListSharedLock@@QAE...	1.000	100% equal
00007	010029ec	?ScIsValidServiceHandle@@YGHPA...	010029ec	?ScIsValidServiceHandle@@YGHPA...	1.000	100% equal

And then, on the Partial Matches tab… we see one, and only one function. This is likely what MS15-050 targeted (the description in the Security Bulletin is that this fixed an "Impersonation Level Check" — the function name sounds like it could be related to an access check!).

IDA View-A | Pseudocode-A | Diff pseudo-code ScStatusAccessCheck - ScStatusAccessCheck | Best matches | Partial matches

Line	Address	Name	Address 2	Name 2	Ratio	Description
00000	01007c95	?ScStatusAccessCheck@@YGKPAU...	01007c84	?ScStatusAccessCheck@@YGKPAU...	0.970	Perfect match, same name

Now that we have our only candidate for the fix delivered in this update, we can investigate what the change actually was. We do this by right-clicking on the function and selecting "Diff pseudo-code". The screenshot below is Diaphora's output:

```
1 MACRO_ERROR_CODE __stdcall ScStatusAccessCheck(DWORD ReturnLength)
2 {
3   struct_ServiceRecord *ServiceRecord; // esi@1
4   HANDLE hThread; // eax@3
5   struct_WPP_GLOBAL_Control **wppControl; // ebx@4
6   LUID *luidToCheck; // eax@7 MAPDST
7   MACRO_ERROR_CODE result; // eax@15 MAPDST
8   TOKEN_STATISTICS TokenInformation; // [sp+8h] [bp-44h]@4
9   LUID systemLuid; // [sp+40h] [bp-Ch]@7
10  HANDLE TokenHandle; // [sp+48h] [bp-4h]@1
11
12  ServiceRecord = (struct_ServiceRecord *)ReturnLength;
13  TokenHandle = 0;
14  if ( !ReturnLength || *(_DWORD *)(ReturnLength + 28) )
15  {
16    result = RpcImpersonateClient(0);
17    if ( result )
18    {
19      ScLogImpersonateFailureEvent(result);
20    }
21    else
22    {
23      hThread = GetCurrentThread();
24      if ( OpenThreadToken(hThread, 8u, 1, &TokenHandle) )
25      {
26        wppControl = &WPP_GLOBAL_Control;
27        if ( GetTokenInformation(TokenHandle, TokenStatistics, &TokenInformation,
28        {
29          systemLuid.HighPart = 0;
30          systemLuid.LowPart = 0x3E7;
31          if ( TokenInformation.TokenType == TokenImpersonation
32            && TokenInformation.ImpersonationLevel < SecurityImpersonation
33            || (ServiceRecord ? (luidToCheck = &ServiceRecord->ImageRecord-
  >AccountLuid) : (luidToCheck = &systemLuid),
34              TokenInformation.AuthenticationId.LowPart != luidToCheck->LowPart
35            || (ServiceRecord ? (luidToCheck = &ServiceRecord->ImageRecord-
  >AccountLuid) : (luidToCheck = &systemLuid),
36              TokenInformation.AuthenticationId.HighPart != luidToCheck->HighPa
37          {
38            ReturnLength = ERROR_ACCESS_DENIED;
39          }
40          else
41          {
42            ReturnLength = NO_ERROR;
43          }
```

```
1 MACRO_ERROR_CODE __stdcall ScStatusAccessCheck(DWORD ReturnLength)
2 {
3   struct_ServiceRecord *ServiceRecord; // esi@1
4   HANDLE hThread; // eax@3
5   struct_WPP_GLOBAL_Control **wppControl; // ebx@4
6   LUID *luidToCheck; // eax@6 MAPDST
7   MACRO_ERROR_CODE result; // eax@14 MAPDST
8   TOKEN_STATISTICS TokenInformation; // [sp+8h] [bp-44h]@4
9   LUID systemLuid; // [sp+40h] [bp-Ch]@5
10  HANDLE TokenHandle; // [sp+48h] [bp-4h]@1
11
12  ServiceRecord = (struct_ServiceRecord *)ReturnLength;
13  TokenHandle = 0;
14  if ( !ReturnLength || *(_DWORD *)(ReturnLength + 28) )
15  {
16    result = RpcImpersonateClient(0);
17    if ( result )
18    {
19      ScLogImpersonateFailureEvent(result);
20    }
21    else
22    {
23      hThread = GetCurrentThread();
24      if ( OpenThreadToken(hThread, 8u, 1, &TokenHandle) )
25      {
26        wppControl = &WPP_GLOBAL_Control;
27        if ( GetTokenInformation(TokenHandle, TokenStatistics, &TokenInformation,
28        {
29          systemLuid.HighPart = 0;
30          systemLuid.LowPart = 0x3E7;
31          if ( ServiceRecord )
32            luidToCheck = &ServiceRecord->ImageRecord->AccountLuid;
33          else
34            luidToCheck = &systemLuid;
35          if ( TokenInformation.AuthenticationId.LowPart != luidToCheck->LowPart
36            || (ServiceRecord ? (luidToCheck = &ServiceRecord->ImageRecord-
  >AccountLuid) : (luidToCheck = &systemLuid),
37              TokenInformation.AuthenticationId.HighPart != luidToCheck->HighPa
38          {
39            ReturnLength = ERROR_ACCESS_DENIED;
40          }
41          else
42          {
43            ReturnLength = NO_ERROR;
44          }
```

Conclusion

At this point, the vulnerability is pretty clear. In at least some cases where an access check is being made due to someone calling the Service Control Manager, the impersonation level isn't verified — meaning that someone with an Anonymous SYSTEM token (for example) could pass off as actually being a SYSTEM caller, and therefore be able to perform actions that only SYSTEM could do. In fact, in this case, we see that the Authentication ID (LUID) of 0x3E7 is checked, which is actually SYSTEM_LUID, making our example reality.

At this point, I won't yet go into the details on which Service Control Manager calls exactly are vulnerable to this incorrect access check (*ScAccessCheck*, which is normally used, actually isn't vulnerable, as it calls *NtAccessCheck*), or how this vulnerability could be used for local privilege escalation, because I wanted to give kudos to Joxean for this amazing plugin and get more people aware of its existence.

Perhaps we'll keep the exploitation for a later post? For some ideas, read up James Forshaw's excellent Project Zero blog post, in which he details another case of poor impersonation checks in the operating system.

This entry was posted on Thursday, May 14th, 2015 at 11:22 am and is filed under Uncategorized. You can follow any responses to this entry through the RSS 2.0 feed. You can leave a response, or trackback from your own site.

3 Responses to "Analyzing MS15-050 With Diaphora"

1. *adi* says:
   May 16, 2015 at 5:17 am

   I'm new to this stuff……so the left version is the correct/patched version, right? because I see that on the left there is a check on impersonation level……
   Thanks!

2. *0xFEEEFEEE* says:
   June 23, 2015 at 3:34 am

   Just curious, what's the tool's name in the last screen shot for using diff?

3. *aionescu* says:
   December 29, 2015 at 1:46 pm

   It's diaphora, the same tool that the blog post talks about 😉

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Submit Comment
