3 How Slow Can You Go?

by James Forshaw

While doing my research into Windows, I tend to find quite a few race condition vulnerabilities. Although these vulnerabilities can be exploited, you typically only get a tiny window of time in which to do it. A fairly typical sequence of actions looks something like this:

1. Do some security check.
2. Access some resource.
3. Perform secure action.

In this case the race condition is between the security check and the action. If we can modify the state of the system in between those actions, it might be possible to elevate privileges or do unexpected things. The time window is typically very small, but if the code is accessing some controllable resource in between the check and the action, we might still be able to create a very reliable exploit.

I wanted to find a way of increasing the time window to win the race in cases where the code accesses a resource we control. The following is an overview of the thought process I went through to come up with a working solution.

3.1 Investigating Object Manager Lookup Performance

Hidden under the hood of Windows NT is the Object Manager Namespace (OMN). You wouldn't typically interact with it directly as the Win32 API for the most part hides it away. The NT kernel defines a set of objects, such as Files, Events, Registry Keys, that can all have a name associated with them. The OMN provides the means to lookup these named objects. It acts like a file system; for example, you can specify a path to an NT system call such as `\BaseNamedObjects\MyEvent`, and an event can be thus looked up.

There are two special object types for use in the OMN: Object Directories and Symbolic Links. Object Directories act as named containers for other objects, whereas Symbolic Links allow a name to be redirected to another OMN path. Symbolic Links are used quite a lot; for example, the Windows drive letters are really symbolic links to the real storage device. When we call an NT system call, the kernel must lookup the entire path, following any symbolic links until it either reaches the named object or fails to find a match.

In this exploit we want to make the process of looking up a resource we control as slow as possible. For example, if we could make it take 1 or 2 seconds, then we've got a massive window of opportunity to win the race condition. Therefore I want to find a way of manipulating the Object Manager lookup process in such a way that we achieve this goal. I am going to present my approach to achieving the required result.

A note about my setup: for my testing I am going to open a named Event object. All testing is done on my 2.8GHz Xeon Workstation. Although it has 20 physical cores, the lookup process won't be parallelized, and therefore that shouldn't be an issue. Xeons tend to have more L2/L3 cache than consumer processors, but if anything this should only make our timings faster. If I can get a long lookup time on my Workstation, it should be possible on pretty much anything else running Windows. Finally, this is all tested on an up-to-date Windows 10; however, not much has changed since Windows 7 that might affect the results.

First let's just measure the time it takes to do a normal lookup. We'll repeat the lookup a $1,000$ times and take the average. The results are probably what we'd expect: the lookup process for a simple named Event is roughly 3μs. That includes the system call transition, lookup process, and the access check on the Event object. Although in theory you could win a race, it seems pretty unlikely, even on a multi-core processor. So let's think about a way of improving the lookup time (and when I say "improve", I mean making the lookup time slower).

An Object Manager path is limited to the maximum string size afforded by the UNICODE_STRING structure.

```
struct UNICODE_STRING {
    USHORT  Length;
    USHORT  MaximumLength;
    PWSTR   Buffer;
}
```

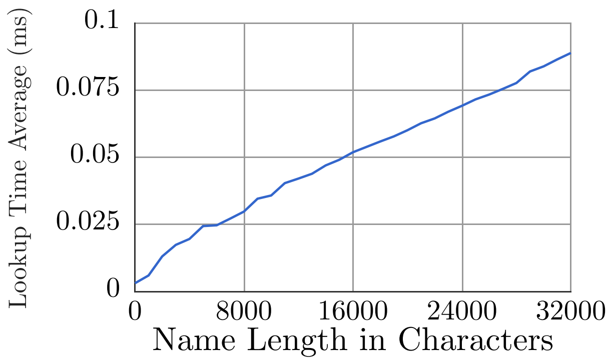
We can see that the Length member is an unsigned 16 bit integer, limiting the maximum length to $2^{16} - 1$. This, however, is a byte count, so in fact this limits us to $2^{15} - 1$ or 32767 characters. From this result, there are two obvious possible approaches we can take:

1. Make a path that contains one very long name. The lookup process would have to compare the entire name using a typical string comparison operation to verify it's accessing the correct object. This should take linear time relative to the length of the string.

2. Make multiple small named directories and repeat. E.g., `\A\A\A\A\...\EventName`. The assumption here is that each lookup takes a fixed amount of time to complete. The operation will again be linear time relative to the depth of recursion of the directories.

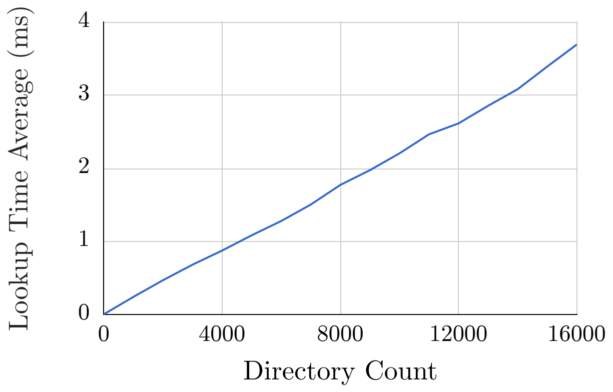
Now it would seem likely that the cost of the entire operation of a single lookup will be worse than a string comparison, a primitive that is typically optimized quite heavily. At this point we have not had to look at any actual kernel code, and we won't start quite yet, so instead empirical testing seems the way to go.

Let's start with the first approach, making a long string and performing a lookup on it. Our name limit is around 32767, although we'll need to be able to make the object in a writable directory such as `\BaseNamedObject`, which reduces the length slightly, but not enough to make significant impact. Therefore, we'll perform the Event opening on names between 1 character and 32,000 characters in length. The results are shown below:



Although this is a little noisy, our assumption of a linear lookup time seems correct. The longer the string, the longer it takes to look it up. For a 32,000 character long string, this seems to top out at roughly 90μs – still not enough in my opinion for a useful primitive, but certainly a start.

Now let's instead look at the recursive directory approach. In this case the upper bound is around 16,000 directories. This is because each path component must contain a backslash and a single character name (i.e. `\A\A\A...`). Therefore our maximum path limit is halved. Of course we'd make the assumption that the time to go through the lookup process is going to be greater than the time it takes to compare 4 Unicode characters, but let's test to make sure. The results are shown below:



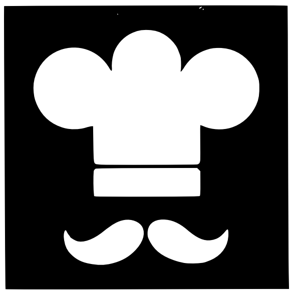
Well, I think that's unequivocal. For 16,000 recursive depth, the average lookup time is around 3700μs, or around 40 times larger than the long path name lookup result. Now, of course, this comes with downsides. For a start, you need to create 16,000 or so directory objects in the kernel. At least on a modern 64 bit Windows this isn't likely to be too taxing, however it's still worth bearing in mind. Also the process must maintain a handle to each of those directories, because otherwise they'd be deleted (as a normal user cannot make kernel objects permanent). Fortunately our handle limit for a single process is of the order of 16 million, so we're a couple of orders of magnitude below the limit of that.

Now, is 3700μs going to be enough for us? Maybe, it's certainly orders of magnitude greater than 3μs. But can we do better? We've now run out of path space, we've filled the absolute maximum allowed string length with recursive directory names. What we could do with is a method of multiplying that effect without requiring a longer path. We can do this by using Object Manager symbolic links. By placing the symbolic link as the last component of the long path we can force the kernel to reparse, and start the lookup all over again. On the final lookup we'll just point the symbolic link to the target.

Ultimately though we can only do this 64 times. Why, can't we do this indefinitely? Well, no—for a fairly obvious reason: each time a symbolic link is encountered the kernel restarts the parsing processes; if you pointed a symbolic link at itself, you'd end up in an infinite loop. The reparse limit of 64 prevents that from becoming a problem. The results are as we expected, the time taken to lookup our event is proportional to both the number of symbolic links and the number of recursive directories. For 64 symbolic links and 16,000 directories it takes approximately 200ms (note I've had to change the order of the result now to milliseconds). At around $\frac{1}{5}$ of a second that should be enough, right? Sure, but I'm greedy; I want more. How can we make the lookup time even worse?

At this point it's time to break out the disassembler and see how the lookup process works under the hood in the kernel. First off, let's see what an object directory structure looks like. We can dump it from a kernel debugging session using WinDBG with the

C&H Best Sellers: Programs That Work!



The Menu][

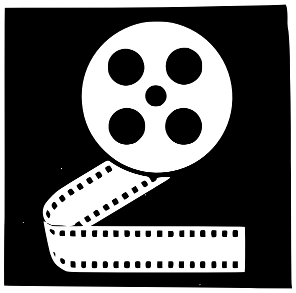
Helps you plan menus and write shopping lists!

NEW!

Plan meals with ease with the all new updated MENU][from C&H. More storage, more information per recipe and other new features in the new version. Operates with 1 disk drive in DOS 3.3.

- **6 meal classifications**
- **2 special counters (calories, sodium, etc.)**
- **24 ingredients per recipe**
- **write menus for 2 week periods**
- **produce printed shopping list**
- **add, change or delete any recipe**
- **24 lines of comments**
- **feed up to 1,295 people per recipe**

Req. 48K Apple, Disk Drive & Printer Applesoft Basic/ Machine Language **$39.95**



The Slide Show

Helps you present a visual, exciting slide demonstration!

High resolution graphics are now more versatile and less expensive than 35MM slides! Create a slide-like presentation on your TV screen with dozens of special effects.

- **educators • sales people • lectures**
- **business • executives • exhibits**
- **free running store displays**
- **cable or closed circuit TV nets**
- **presentations**

Req. 48K Apple, Disk Drive. In Applesoft Basic/ Machine Language **$49.95**

See your favorite APPLE dealer or order direct. Send check or money order to:

C & H VIDEO
Box 201 • Hummelstown PA 17036
Specify DOS 3.2 or 3.3 PA Res. Add 6% sales tax

For charges call
717-533-8480
Between 9am to 9pm

VISA master charge

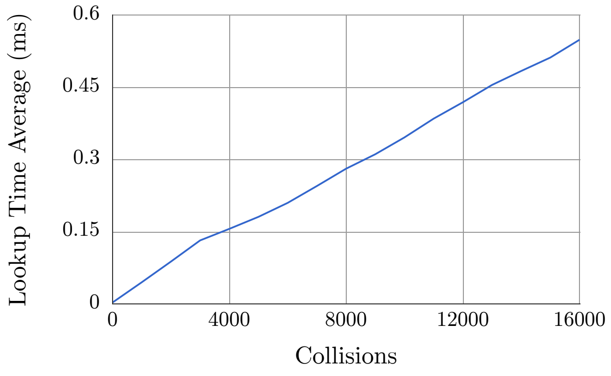
command `dt nt!_OBJECT_DIRECTORY`. Converted back to a C-style structure, it looks something like the following:

```
struct OBJECT_DIRECTORY
{
    POBJECT_DIRECTORY_ENTRY HashBuckets[37];
    EX_PUSH_LOCK Lock;
    PDEVICE_MAP DeviceMap;
    ULONG SessionId;
    PVOID NamespaceEntry;
    ULONG Flags;
    POBJECT_DIRECTORY ShadowDirectory;
}
```

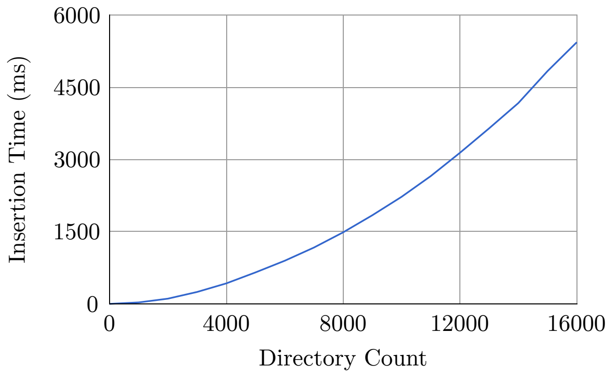
Based on the presence of the HashBucket field, it's safe to assume that the kernel is using a hash table to store directory entries. This makes some sense, because if the kernel just maintained a list of directory entries, this would be pretty poor for performance. With a hash table the lookup time is much reduced as long as the hashing algorithm does a good job of reducing collisions. This is only the case though if the algorithm isn't being actively exploited. As we're trying to increase the cost of lookups, we can intentionally add entries with collisions to make the lookup process take the worst case time, which is linear relative to the number of entries in a directory. This again provides us with another scaling factor, and in this case the number of entries is only going to be limited by available memory, as we are never going to need to put the name into the path.

So what's the algorithm for the hash? The main function of interest is `ObpLookupObjectName`, which is referenced by functions such as `ObReferenceObjectByName`. The directory entry logic is buried somewhere in this large function; however, fortunately there's a helper function `ObpLookupDirectoryEntryEx`, which has the same logic (it isn't actually called by `ObpLookupObjectName`, but it doesn't matter) that is smaller and easier to reverse (Figure 10).

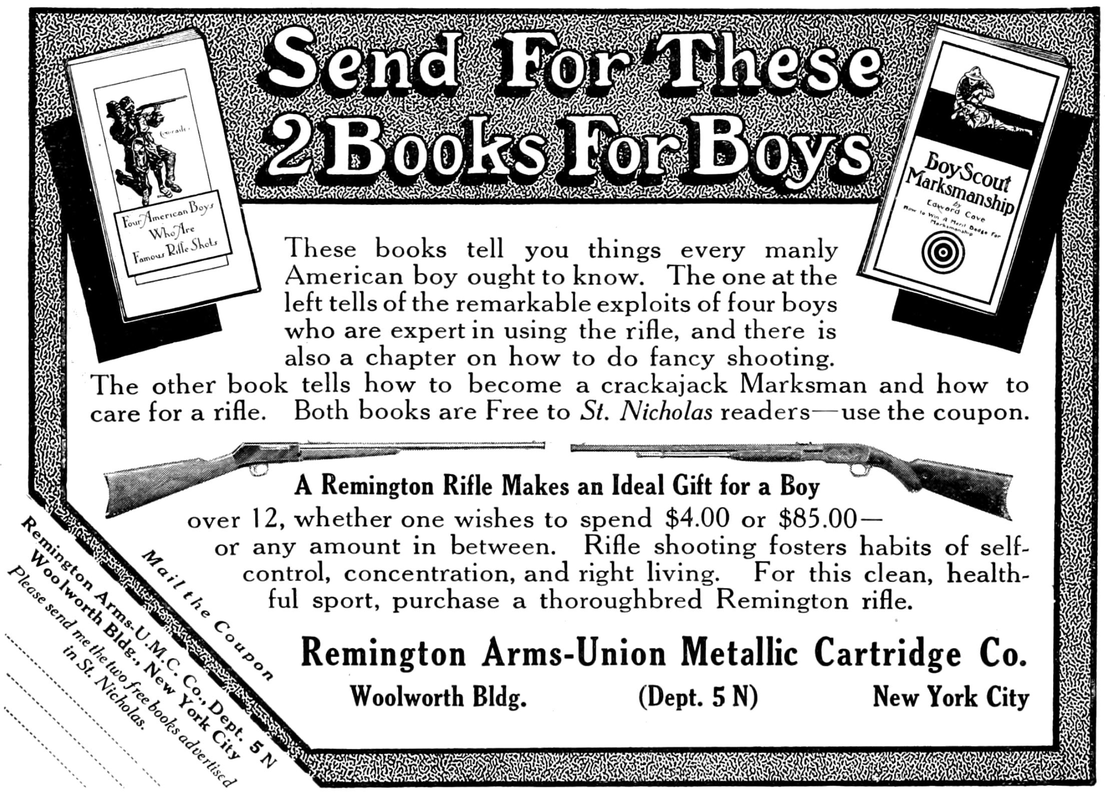
So the hashing algorithm is pretty simple; it repeatedly mixes the bits of the current hash value and then adds the uppercase Unicode character to the hash. We could work out a clever way of getting hash collisions from this, but actually it's pretty simple. The object manager allows us to specify names containing NULL characters, therefore if we take our target name, say 'A', and prefix it with increasing length strings containing only NULL, we get both Hash and Bucket collisions. This does limit us to creating only 32,000 or so colliding entries before we run out of strings to create them, but, as we'll see in a minute, that's not a problem. Let's look at the results of doing this for a single directory:



Yet again, a nice linear graph. For a given collision count it's nowhere near as good as the recursive directory approach, but it is a multiplicative factor in the lookup time, which we can abuse. So you'd think we can now easily apply this to all our 16,000 recursive directories, add in symbolic links, and probably get an insane lookup time. Yes, we would, however there's a problem, insertion time. Every time we add a new entry to a directory, the kernel must do a lookup to check that the entry doesn't already exist. This means that, for every entry we add, we must do $(n-1)^2$ checks in the hash bucket just to find that we don't have the entry before we insert it. This means that the time to add a new entry is approximately proportional to the square of the number of entries. Sure it's not a cubic or exponential increase, but that's hardly a consolation. To prove that this is the case we can just measure the insertion time:



That graph shows a pretty clear n^2 trend for the insertion time. If, say, we wanted to create a directory entry with 16,000 collisions, it takes close to 5.5 seconds. If we wanted to then do that for all 16,000



Send For These 2 Books For Boys

These books tell you things every manly American boy ought to know. The one at the left tells of the remarkable exploits of four boys who are expert in using the rifle, and there is also a chapter on how to do fancy shooting. The other book tells how to become a crackajack Marksman and how to care for a rifle. Both books are Free to *St. Nicholas* readers—use the coupon.

A Remington Rifle Makes an Ideal Gift for a Boy over 12, whether one wishes to spend $4.00 or $85.00—or any amount in between. Rifle shooting fosters habits of self-control, concentration, and right living. For this clean, healthful sport, purchase a thoroughbred Remington rifle.

Remington Arms-Union Metallic Cartridge Co.
Woolworth Bldg. (Dept. 5 N) New York City

Mail the Coupon

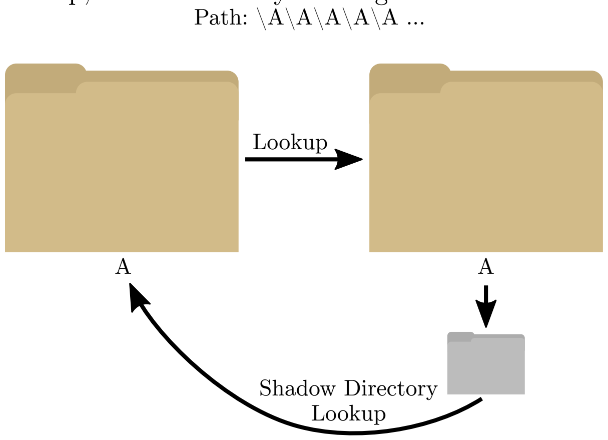
Remington Arms-U.M.C. Co., Dept. 5 N
Woolworth Bldg., New York City
Please send me the two free books advertised in St. Nicholas.

```
POBJECT_DIRECTORY ObpLookupDirectoryEntryEx(POBJECT_DIRECTORY Directory,
                                            PUNICODE_STRING Name,
                                            ULONG AttributeFlags) {
  BOOLEAN CaseInSensitive = (AttributeFlags & OBJ_CASE_INSENSITIVE) != 0;
  SIZE_T CharCount = Name->Length / sizeof(WCHAR);
  WCHAR* Buffer = Name->Buffer;
  ULONG Hash = 0;
  while (CharCount) {
    Hash = (Hash / 2) + 3 * Hash;
    Hash += RtlUpcaseUnicodeChar(*Buffer);
    Buffer++;
    CharCount--;
  }

  OBJECT_DIRECTORY_ENTRY* Entry = Directory->HashBuckets[Hash % 37];
  while(Entry) {
    if (Entry->HashValue == Hash) {
      if (RtlEqualUnicodeString(Name,
            ObpGetObjectName(Entry->Object), CaseInSensitive)) {
        ObReferenceObject(Entry->Object);
        return Entry->Object;
      }
    }
    Entry = Entry->ChainLink;
  }

  return NULL;
}
```

Figure 10. `ObpLookupDirectoryEntryEx()`

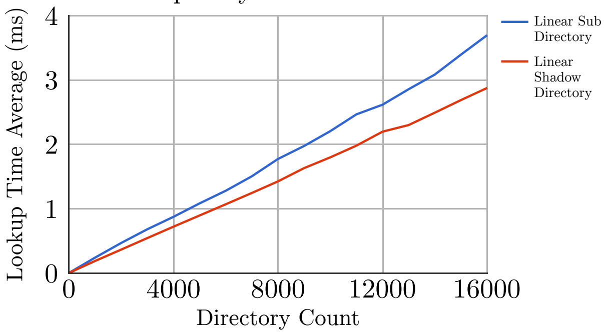
recursive directory entries, it would take around 24 hours! Now, I think we're going a bit over the top here, and by fiddling with the values we can get something that doesn't take too long to set up and gives us a long lookup time. But I'm still greedy; I want to see how far I can push the lookup time. Is there any way we can get the best of all worlds?

The final piece of the puzzle is to bring in Shadow directories, which allow the Object Manager a fallback path if it can't find an entry in a directory. You can use almost any other Object Manager directory as a shadow, which will allow us to control the lookup behavior. A Shadow Directory has a crucial difference from symbolic links, as it doesn't cause a reparse to occur in the lookup process. This means they're not restricted to the 64 reparse limit. As each lookup consumes a path component, eventually there will be no more paths to lookup. If we put together two directories in the following arrangement, we can pass a similar path to our recursive directory lookup, without actually creating all the directories.



So how does this actually work? If we open a path of the form `\A\A\A\A\A...`, the kernel will first lookup the initial 'A' directory. This is the directory on the left of the diagram. It will then try to open the next 'A' directory, which is on the right, which again it will find. Next the kernel again looks up 'A', but in this case it doesn't exist. As the directory has a shadow link to its parent, it looks there instead, finds the same 'A' directory, and repeats the process. This will continue until we run out of path elements to lookup.

So let's determine the performance of this approach. We'd perhaps expect it to be less performant relative to actually creating all those directories if only because of the cache effects of the processor. But hopefully it won't be too far behind.



Looks good. Yes, the performance is lower than actually creating the directories, but once we bring collisions into the mix, that's not really going to matter much. So the final result is that instead of creating 16,000 directories with 16,000 collisions we can do it with just 2 directories, which is far more manageable and only takes around 11 seconds on my workstation. So, to sign off, let's combine everything together.

1. 16,000 path components using 2 object directories in a shadow configuration

2. 16,000 collisions per directory

3. 64 symbolic link reparses

And the resulting time for a single lookup on my workstation is **drum roll please** 19 minutes! I think we might just be able to win the race condition with that.

Code examples can be found attached to this document.[10]

3.2 Conclusion

So after all that effort we can make the kernel take around 19 minutes to lookup a single controlled resource path. That's pretty impressive. We have many options to get the kernel to start the lookup process, allowing us to use not just files and registry keys but almost any named event. It's a typical tale of unexpected behavior when facing pathological input, and it's not really surprising Microsoft wouldn't optimize for this use case.

[10] `unzip pocorgtfo13.pdf object_manager_lookup_poc.cs`