How Malware Generates Mutex Names to Evade Detection

By SANS Internet Storm Center

Archived: 2026-04-05 21:20:05 UTC

Malicious software sometimes uses mutex objects to avoid infecting the system more than once, as well as to coordinate communications among its multiple components on the host. Incident responders can look for known mutex names to spot the presence of malware on the system. To evade detection, some malware avoids using a hardcoded name for its mutex, as is the case with the specimen discussed in this note.

Static Mutex Names as Indicators of Compromise

For background details about mutex (a.k.a. mutant) values and their role in incident response, see my earlier article Looking at Mutex Objects for Malware Discovery and Indicators of Compromise. As I described there, when examining a potentially-infected system, we can look for names of mutex objects known to be used by malicious programs. Moreover, in some circumstances, mutex values could be used as markers to immunize the system from infection. Not all malware uses mutex objects, but this is an indicator that's worth looking for.

This approach works well when you've already found malware on some systems and examined it to determine that it uses specific mutex names. You can also obtain mutex name details from threat intelligence feeds, sites such as TotalHash, and the use of automated malware analysis tools. For a long list of mutex names used by various malware samples, take a look at the list published by Hexacorn.

However, malware could name its mutex objects in a less-predictable way, as discussed below.

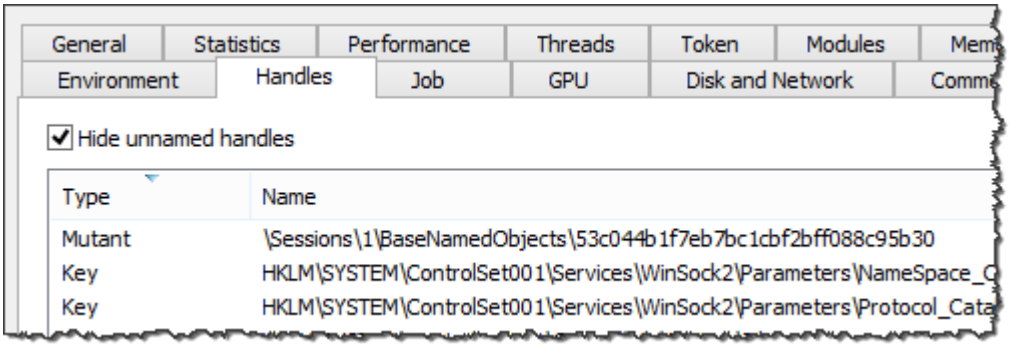
Mutex Defined by TreasureHunter Malware

Let's take a brief look at a malware sample with the MD5 hash 070e9a317ee53ac3814eb86bc7d5bf49. Its author called this malicious program TreasureHunter, according to a couple of strings that were embedded into the file:

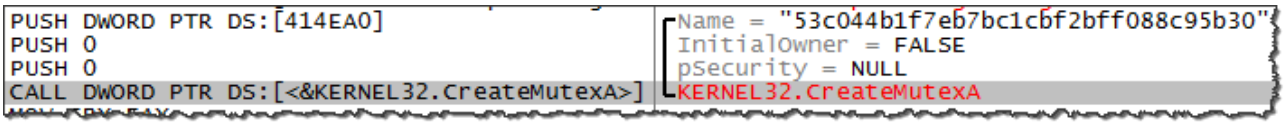
```
c:\users\admin\documents\visual studio 2012\projects\treasurehunter\release\treasurehunter.pdb
TreasureHunter version 0.1 Alpha
```

This malicious file was named jucheck.exe and was discovered under the %AppData% folder. We won't look at all aspects of this malicious program, since my goal is to focus on its use of mutex objects for this article.

If you infect a laboratory system with TreasureHunter, you can use a tool such as Process Hacker to spot that this malicious process creates a mutex named 53c044b1f7eb7bc1cbf2bff088c95b30.



You can further investigate TreasureHunter's use of the mutex by running the malicious executable in a debugger such as OllyDbg. As you can see in the screenshot below, specimen calls CreateMutexA to create the mutex:

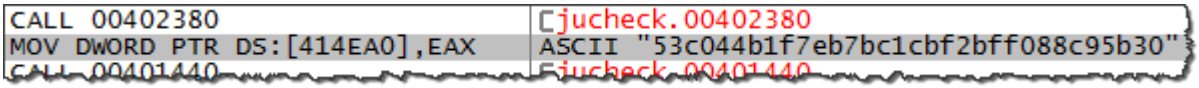


Running the specimen multiple times results in the same mutex name being used across the experiments. At this point, you might be inclined to use the mutex name 53c044b1f7eb7bc1cbf2bff088c95b30 as an indicator of compromise for TreasureHunter. However, further analysis would show that on other systems this sample names mutex objects differently. For instance, VirusTotal reports the mutex name 3ed1ed60c7d7374bf0dd76fc664b39cd, while VxStream Sandbox reports the name 3ac22cadc45e0558cad697d777f6c3d3. Why such a discrepancy?

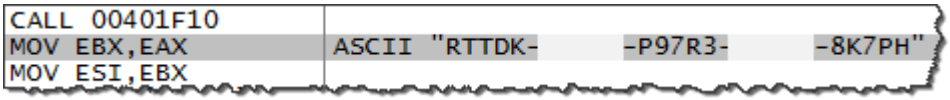
How TreasureHunter Generates the Name of Its Mutex

The names of this specimen's mutex object appear to be MD5 hashes, but what value is the specimen hashing? Looking more closely at TreasureHunter with the help of a debugger can shed some light on this functionality.

The OllyDbg screenshot above shows that the specimen stored the name of its mutex in a variable located at the offset 414EA0. You could scroll up in the program's code to find which assembly instruction saves data into that memory location. As shown in the following screenshot, the string representing the name of the mutex is being moved into that location from the register EAX. This register stores the results of the previously-invoked function, which in our example is 402480:



We could use the debugger to drill into that function to determine how it derives the name of the mutex. That function ends up calling function 401F10, which returns the string RTTDK-XXXXX-P97R3-YYYYYY-8K7PH, as shown below. I replaced some characters of the string with XXXXX and YYYYY for privacy reasons:



Later, the code calls function 401020, which returns the string RTTDK-XXXXX-P97R3-YYYYYY-8K7PH(as shown below:

```
CALL 00401020          jucheck.00401020
PUSH EBX               Arg1
MOV EDI,EAX            ASCII "RTTDK-      -P97R3-      -8K7PH("
CALL 00404BDF          jucheck.00404BDF
```

Further, the code computes the MD5 hash of the string RTTDK-XXXXX-P97R3-YYYYYY-8K7PH(to derive 53c044b1f7eb7bc1cbf2bff088c95b30, which it ends up using as the mutex name.

Where does the string RTTDK-XXXXX-P97R3-YYYYY-8K7PH come from? To figure that out, we can drill into function 401F10, which returned this value. That function invokes several API calls to read the system's registry:

```
PUSH EAX               pResult = 13E68C -> 0013E6DC
PUSH 1                 DesiredAccess = KEY_QUERY_VALUE
PUSH 0                 Reserved = 0
PUSH OFFSET 00410108   SubKey = "SOFTWARE\Microsoft\Windows NT\CurrentVersion"
PUSH 80000002          hKey = HKEY_LOCAL_MACHINE
CALL ESI               ADVAPI32.RegOpenKeyExW
```

The goal of this code is to read HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\DigitalProductId:

```
PUSH EAX                                pDataLen => OFFSET LOCAL.1562
LEA EAX,[LOCAL.520]
PUSH EAX                                pData => OFFSET LOCAL.520
PUSH 0                                  pType = NULL
PUSH 0                                  Reserved = 0
PUSH OFFSET 00410164                    Name = "DigitalProductId"
PUSH DWORD PTR SS:[LOCAL.1561]          hKey => [LOCAL.1561]
CALL EDI                                ADVAPI32.RegQueryValueExW
```

This is where Windows stores its Product ID. The specimen also looked in several other registry locations where Windows sometimes stores this information.

Further in the function, the specimen's code transformed the Product ID's into the RTTDK-XXXXX-P97R3-YYYYYY-8K7PH that it later used as the basis for its mutex name using a deterministic algorithm that I had neither the patience, nor reason to reverse-engineer. This is why when this specimen ran in automated malware analysis sandboxes it used a different mutex value--those products must have had different Windows Product IDs.

The Use of Non-Static Mutex Values in Malicious Software

Malware authors who wish to employ mutex objects need a predictable way of naming those objects, so that multiple instances of malicious code running on the infected host can refer to the same mutex. A typical way to accomplish this has been to hard-code the name of the mutex. The author of TreasureHunter decided to use a more sophisticated approach of deriving the name of the mutex based on the system's Product ID. This helped the specimen evade detection in situations where incident responders or anti-malware tools attempted to use a static object name as the indicator of compromise.

-- Lenny Zeltser

Lenny Zeltser focuses on safeguarding customers' IT operations at NCR Corp. He also teaches how to analyze malware at SANS Institute. Lenny is active on Twitter and Google+. He also writes a security blog, where he contemplated using mutex objects and other infection markers for immunizing systems.

Source: https://isc.sans.edu/diary/How+Malware+Generates+Mutex+Names+to+Evade+Detection/19429/