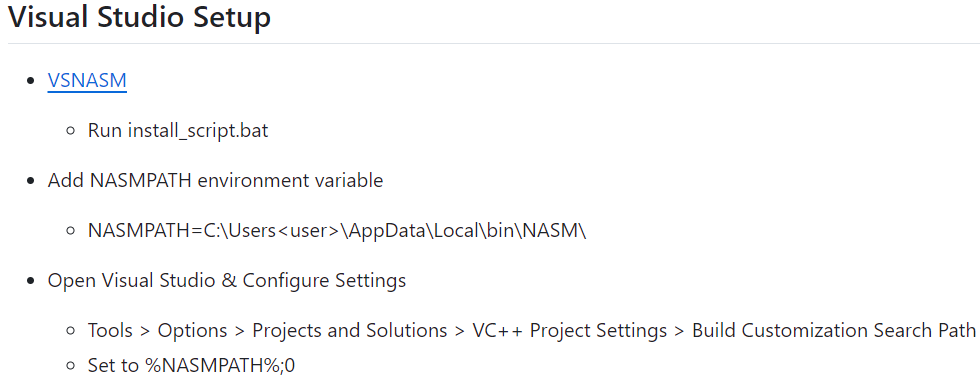
# Cronos Research

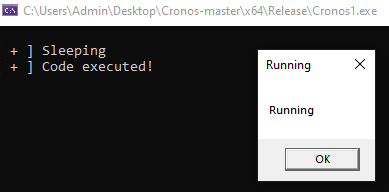
## Compiling cronos

In order to compile cronos I followed the following instructions:

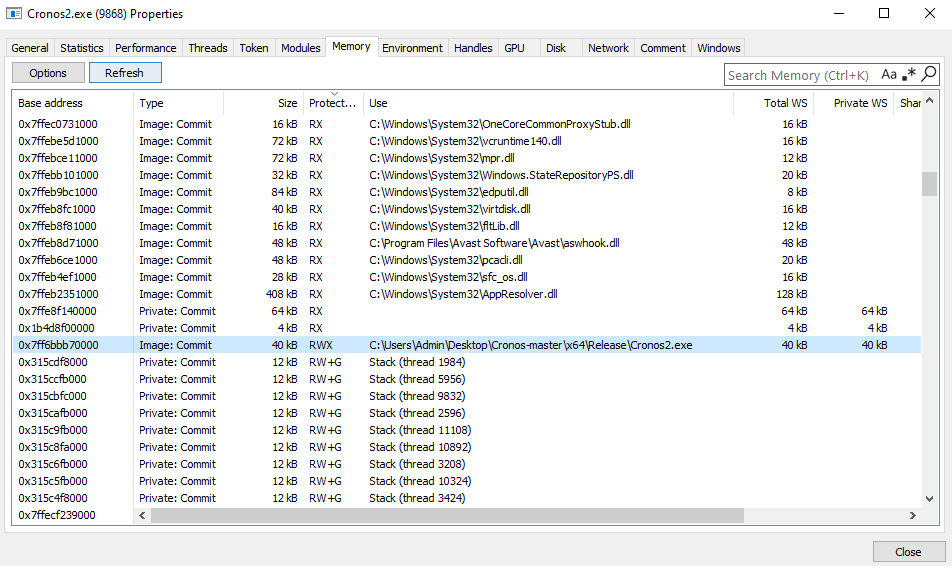


(installing nasm, adding it to the path system variable and to the visual studio search path)

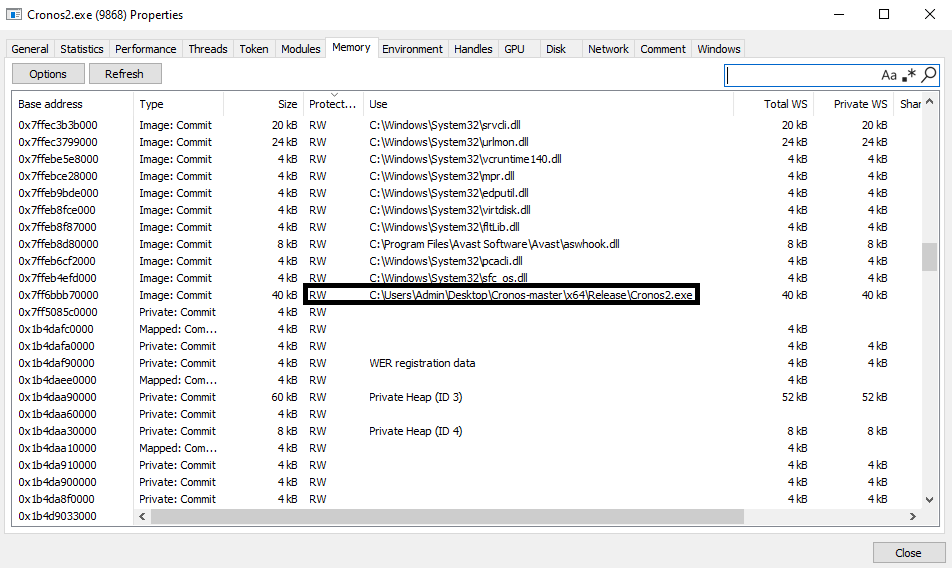
## Cronos running



You can see that while running the main executable’s memory region marked as RWX:



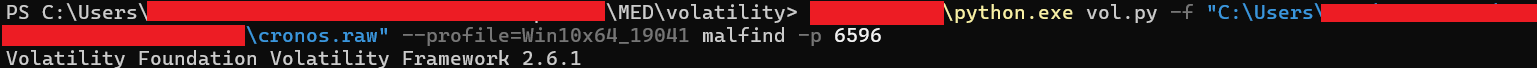
And while it sleeps it removes the executable permission:



## Cronos against malfind

I ran the compiled cronos.exe, and I took a raw memory dump using winpmem.

Then I ran volatility’s malfind to see if it finds cronos in memory:



No result!  
**Cronos bypasses malfind successfully!**

## How does it work?

Cronos mechanism - high level:

1. Changing the image's protection to RW.
2. Encrypt the image.
3. Decrypt the image.
4. Add execution privileges to the image.

How?  
Everything is built on top of the undocumented NtContinue function, which is a function that gets as parameter a thread context, and use it to continue the thread’s execution.  
You can see it here: [NTAPI Undocumented Functions](http://undocumented.ntinternals.net/index.html?page=UserMode%2FUndocumented%20Functions%2FNT%20Objects%2FThread%2FNtContinue.html)

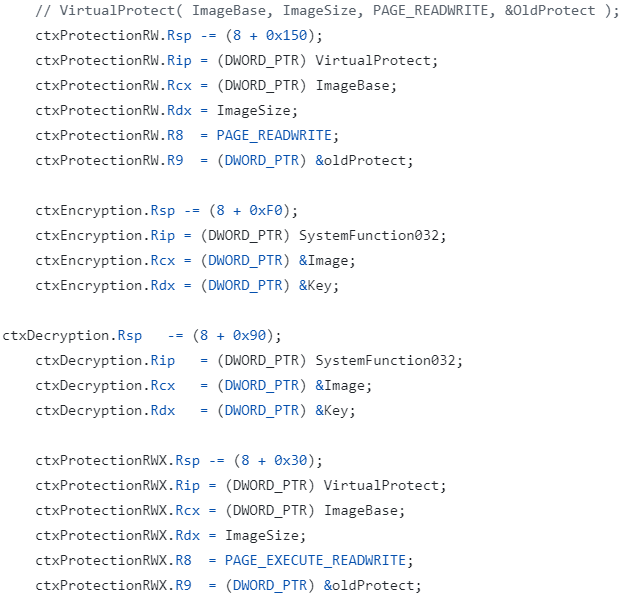
For the encryption and decryption it uses the undocumented function SystemFunction032 that gets a buffer and a key, and encrypts\decrypts the data using RC4.  
More details: [Encrypting Shellcode using SystemFunction032/033 | 🔐Blog of Osanda](https://osandamalith.com/2022/11/10/encrypting-shellcode-using-systemfunction032-033/)

For the protections changes (RW\RWX) it uses the traditional VirtualProtect function.

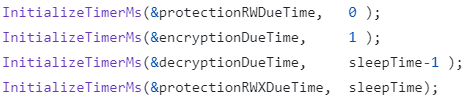
So, for every one of the above four stages, it builds a thread context and set a waitable timer that its completion routine is NtContinue. The parameter of the completion routine is the thread context, and the time of execution is according to the operation (change to RW and encrypt are done immediately, and the decryption + change to RWX are done in the end of the sleep time).

Code screenshots:

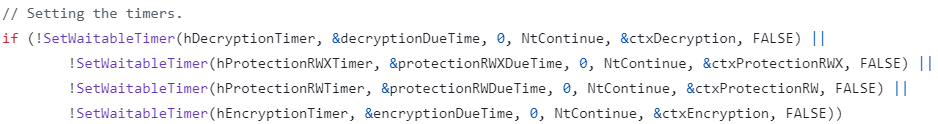
Building thread contexts per operation:



Setting time of execution per operation:



Setting the timers:



## Cronos Detection

How can we detect it?

Whether it’s surprising or not, its core functionality and IOCs are pretty similar to gargoyle, which we already implemented a method to catch it in our POC. It’s just a little bit more sophisticated.

Just for reminder, in Gargoyle we parsed the kernel’s timers list in order to find all waitable timers, and then using unicorn emulator we followed the completion routine to see if it calls VirtualProtect (setting a memory region executable).  
The address of the region which becomes executable is suspected to contain a Gargoyle payload.

Here we can do something similar, but we need to make some adjustments.  
Actually, the already existing algorithm finds some payloads in cronos.exe (which surprised me a little bit):



But it locates the payload somewhere in ntdll.dll (which is definitely not the right place), and the reason is probably that we assume that the payload’s address resides on the stack:



And it’s apparently not the calling convention here (as it’s 64 bit).

**Therefore, in order to detect cronos we can do the following:**

1. Adhere to our already implemented algorithm and make the adjustments so that it would extract VirtualProtect’s parameters correctly. The algorithm itself should be valid for this method as well.
2. Real time detection using ETW subscription is also valid for this method, as it uses VirtualProtect in the same way in the end of the day.,
3. Follow calls of the undocumented functions used by cronos: SystemFunction032\033 and NtContinue, and detect cronos this way (SystemFunction032\033 get the payload’s address as parameter so it’s useful too).  
   It’s a little bit discouraged because it’s too much implementation-independent (these functions add extended features and are not in the core mechanism as VirtualProtect and SetWaitableTimer).