Recognition of Excellence at Null Byte (Fellows, Awards, Recommendations, & Certifications)
How to Post to Null Byte [4.10.15 Revision]
4 Ways to Crack a Facebook Password and How to Protect Yourself from Them
7 Android-Only Apps That Will Make iPhone Users Green with Envy
Why YOU Should Study Digital Forensics
Social Enginee Total Guide

A Guide on Runtime Crypters

Posted By dontrustme 396 yesterday

What's good, peeps? I've been noticing some rather advanced tutorials slowly emerging here on Null Byte and I know that people want more of them but I've been reluctant to post something of such caliber because I fear that the information will just go over their heads, but hey, as long as it's there, people can always go off to research themselves and eventually understand. So here is my contribution to the gradual and inevitable progression of Null Byte!

Welcome, readers, to a guide and walkthrough on runtime crypters! This section will specifically cover Windows systems so if you're a hater and cringe at the slightest sound of Microsoft's grotesque baby, feel free to close this tab, delete your history, exit your browser, pour some oil onto your machine and sanitize it with fire! Else, please continue onwards. If you do not know what a crypter is, please proceed onto this article. Also, please bare with the theory, I know that it's boring and I apologize for that.

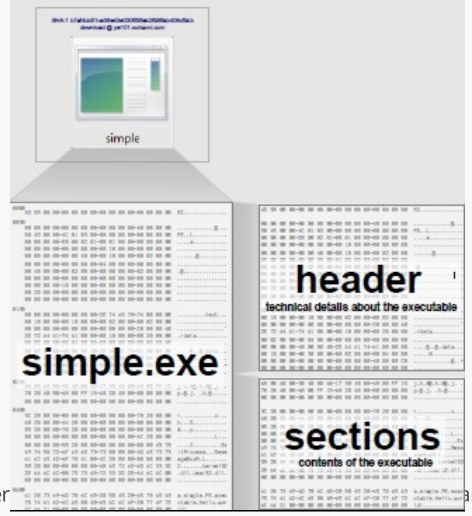
The following article details the internals of the *Portable Executable* (PE) format and some concepts of Windows' memory management. I will only be covering the relevant information so if you wish to understand more, you will need to do some more research.

Note: I am still currently learning more about this so if I happen to get any information incorrect, please leave a comment below or drop a message in my inbox and I will try to patch it ASAP.

Disclaimer: This is an article which shows how the runtime crypter works. It is meant to be a guide, not a tutorial so not every little piece of information will be provided.

Introduction to the Portable Executable

The PE file format is a Windows file format which is based upon and is a modification to Unix's *Common Object File Format* (COFF), a specification for an executable, object or shared library file. It originated as a result of the evolution into Windows NT and Windows 95 systems which allows the operating system to be able to more efficiently load programs into memory as opposed to the implementations in DOS. Let's take a look into a high-level overview representation of a PE file. The following images are extracts of *PE 101 - A Windows Executable Walkthrough* and *The PE Format* by Ange Albertini, pe101.corkami.com, corkami.blogspot.com.

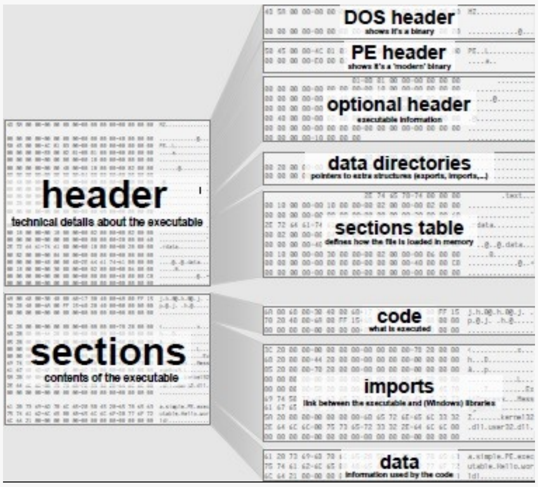


We can see her … arts: the header and sections.

The header contains crucial information about the file such as its overall management (locations and sizes) and how it should be handled in memory.

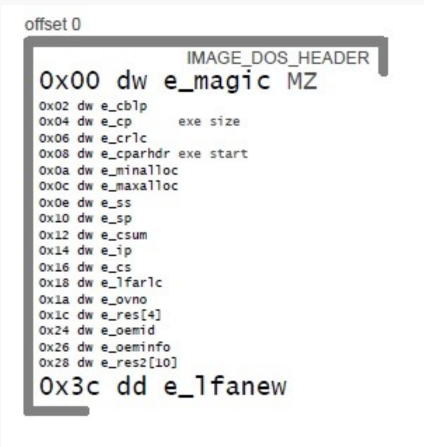
The sections are where the actual data resides such as the code, data and resources (icons, images, GUI, etc.) - those who have been following my C tutorials should be somewhat familiar with this.

We'll now take a deeper look into these two segments.



The PE Headers

Let's cover the headers first.

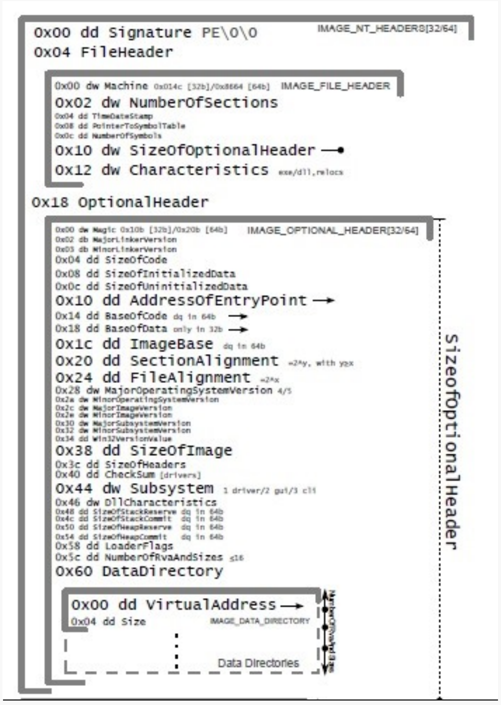


The DOS header contains a lot of values but since this article will only be focusing on the important details, there are only two members which are of use. The *e_magic* member is also known as the DOS signature states that the particular file is a binary file. The signature is 'MZ' or 0x5A4D (little endian) stands for Mark Zbikowski who was one of the developers of MS-DOS.

8

KUDOS

What usually comes after the DOS header is the DOS stub program. We cannot see it here but it's there (trust me). What happens when an executable is run in a DOS environment is that it will execute the DOS stub program. Normally, it will just display the message "*This program cannot be run in DOS mode*". I'm sure you've seen this before if you've opened up an executable in Notepad. Of course, the DOS stub can be modified to contain a fully-fledged DOS program or whatever you wish. Because of this, the size of it can be variable and as a result, the last member of the DOS header, *e_lfanew*, points to the beginning of the PE header using what we call a *relative offset*. What is it relative to? The beginning of the executable image, AKA, the *ImageBase* when in memory. We'll look at this when we reach it.



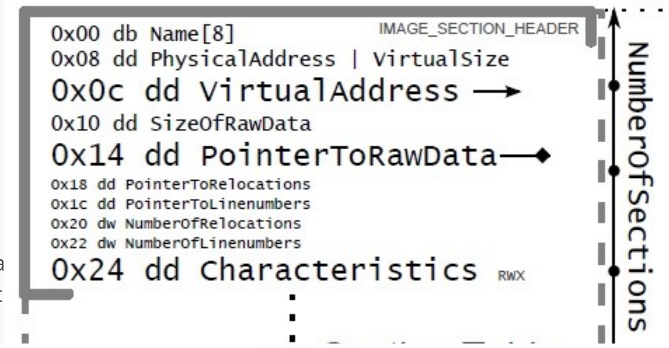
The PE header or otherwise known as the NT header contains two smaller headers inside but in this image, we can only see one: the *Optional header*. The other is called the *File header* which comes directly from the COFF. The first member of the PE header is the PE signature which has the value of 'PE\0\0', i.e. "PE" with two null bytes and represents a PE file.

In the File header, we can see a member called *NumberOfSections* which contains the number of sections the binary file has. We can also see *Characteristics* which states whether the binary file is an EXE, DLL, etc.

In the Optional header, there are the *AddressOfEntryPoint*, *ImageBase*, *SizeOfImage* and *SizeOfHeaders* members. The *AddressOfEntryPoint* is the relative offset (again!) to the entry point, AKA, where the code begins execution. The *ImageBase* member describes the address location *in memory* of where the entire PE file should be loaded from (the beginning) and this value is usually 0x00400000. All relative offsets or *Relative Virtual Address* (RVA) are calculated from this point. The last interesting member is the *SizeOfImage* which contains the size of the entire executable image, i.e. the size of the headers + the size of the sections in memory. If we just want the size of the headers (all the headers combined), the *SizeOfHeaders* will provide that data for us.

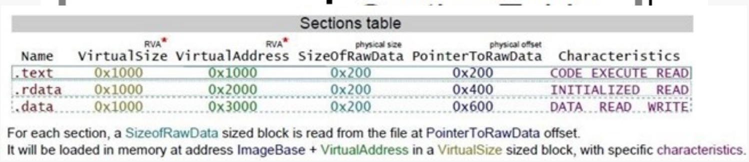
We'll skip over the data directories part since it isn't important to us. Briefly, the data directories contains the information about the locations of Import and Export tables, Resources, Certificates, Copyright information, etc.

Fina



the

sect



Sections table

Name	VirtualSize (RVA*)	VirtualAddress (RVA*)	SizeOfRawData (physical size)	PointerToRawData (physical offset)	Characteristics
.text	0x1000	0x1000	0x200	0x200	CODE EXECUTE READ
.rdata	0x1000	0x2000	0x200	0x400	INITIALIZED READ
.data	0x1000	0x3000	0x200	0x600	DATA READ WRITE

For each section, a SizeofRawData sized block is read from the file at PointerToRawData offset.
It will be loaded in memory at address ImageBase + VirtualAddress in a VirtualSize sized block, with specific characteristics.

The *Name* member contains the name of the section (.text, .data, .rdata, .idata, .edata, .CRT, .tls, etc.) with a maximum of eight characters **not** including a null terminating byte (for those who do C/++ or other similar languages).

The *VirtualSize* member is the size of the section in memory and the *VirtualAddress* is the RVA of the section (from *ImageBase*). The *SizeOfRawData* and *PointerToRawData* are members which detail the physical data of the file on disk. The *SizeOfRawData* is the size of the section and the *PointerToRawData* is the file offset to the beginning of the section (from the beginning of the file). The *Characteristics* of each section states how each section should be handled, whether it's executable or read-only, etc.

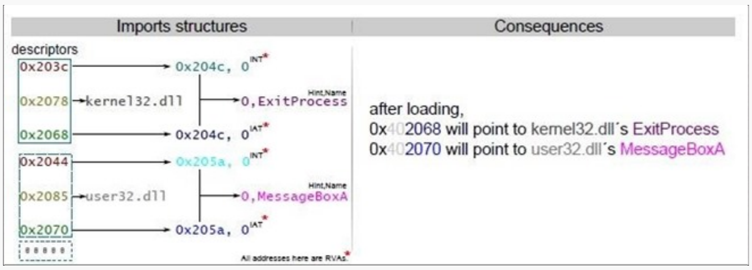
The PE Sections

The sections in the PE file are where the actual information is stored. Let's cover some common sections: *.text*, *.data*, *.rdata*, *.idata*.

The *.text* section (or *code* for Borland) is where the executable code exists and is given the *Characteristics CODE*, *EXECUTE* and *READ*. We want it to be executable and read-only as we do not want anything to corrupt the data.

The *.data* section is where the global data is stored... Yep, that's pretty much it... The *.rdata* is the read-only part of *.data*, i.e. constants exist here such as strings.

Lastly, the *.idata* is where the imports are, AKA, the imported functions for the executable. This also includes the respective DLL names from which the functions are imported.



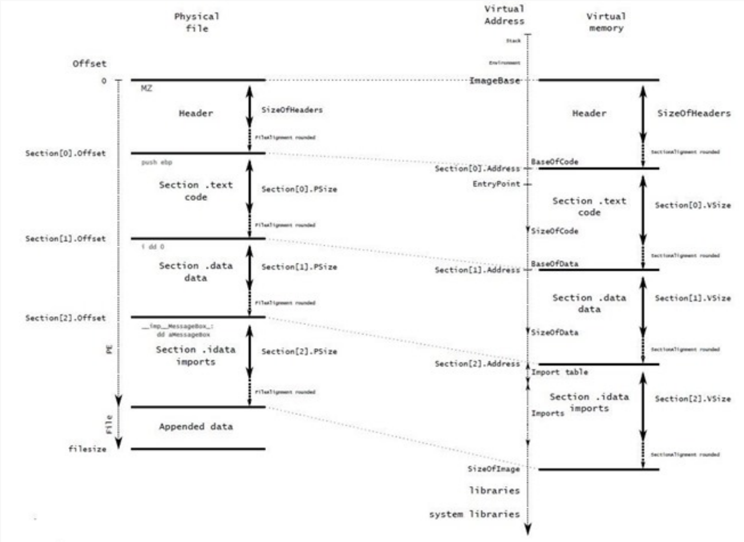
As we can see above, the imported functions *ExitProcess* and *MessageBox* are imported from the DLLs *kernel32* and *user32* respectively. For the C standard library, imported functions such as *printf* and *malloc* come from the *msvcrt* (Microsoft Visual C++ Run Time) library.

PE Format Conclusion

That's it for the introduction to the PE format. As you could see, there was a LOT going on so if you want to learn more, you'll have to research yourself.

Intermezzo

Before we can actually detail how runtime crypters work, we need to know how Windows loads PE files into memory. Here is basically what it looks like.



When Windows maps an executable file into memory, it's pretty much the same thing as on disk except that the sections will be split up into their own parts in memory accordingly. The information for this lies in the respective section's section header.

Now let's look at how runtime crypters do their magic...

Unraveling the Mysteries of the Runtime Crypter

Finally, we have reached the interesting part of the article! Woo! Let's not waste any time and dive down the rabbit hole!

A method that runtime crypters can use is called the *Run PE* or *Dynamic Forking* process. What this means is that our crypter will hold the information of the obfuscated PE file which will be set up in memory and then executed as a process.

Step 1: Creating the Process

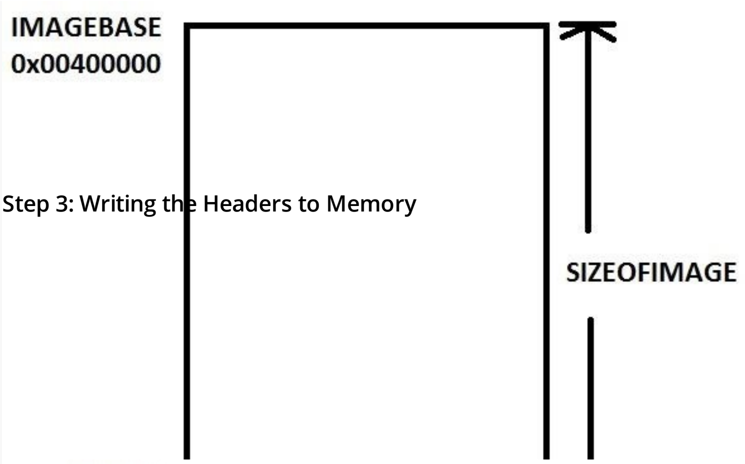
We'll need to create the process first to get a handle to the process and thread of our soon-to-be malware (or not, whatever you want). We'll do this using the CreateProcess function with the *CREATE_SUSPENDED* flag to suspend the thread.

Step 2: Allocating Virtual Memory Space

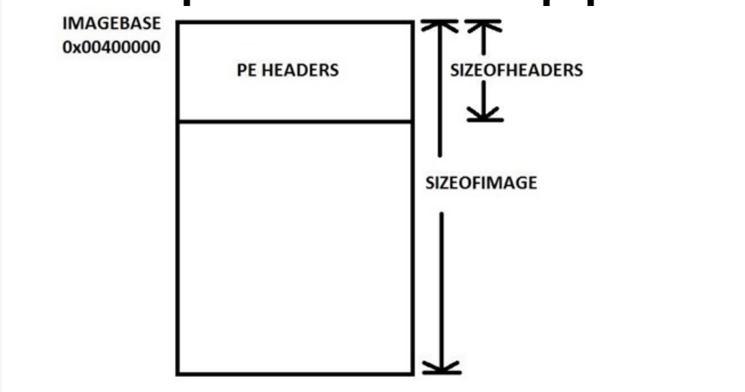
We'll require the memory space large enough to hold our entire executable image but how do we know how big it is? If we look back a bit on this article, I've stated that the *SizeOfImage* member of the Optional header stores the size of the memory space required. Starting from the base address of Optional header's *ImageBase* member, we use VirtualAllocEx to allocate the required space for us. Here's what our memory space should look like:

8

KUDOS

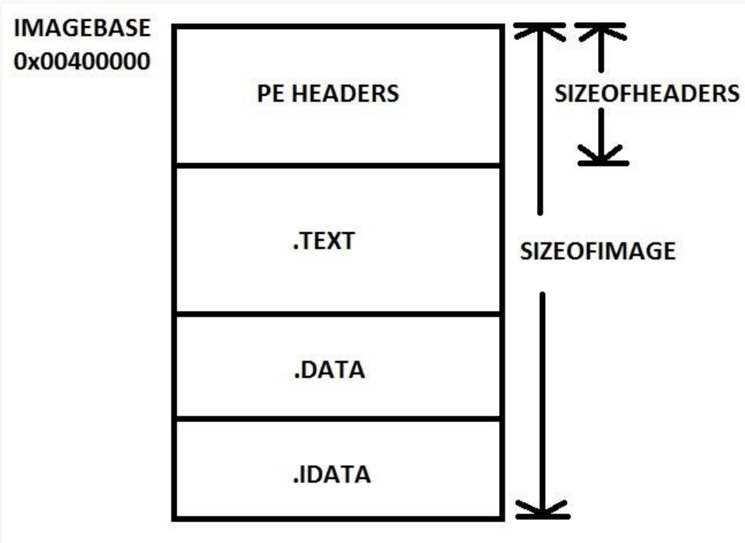


Step 3: Writing the Headers to Memory



Step 4: Writing the Sections into Memory

All we need now is to write the sections into memory. Using WriteProcessMemory again, we'll perform this task using the information from the section headers. This time, instead of starting from *ImageBase*, we'll need to calculate the section's beginning virtual address (Section header's *VirtualAddress* member) and write until all the data has been written using the Section header's *SizeOfRawData* member. Remember, we need to do this for **all** of the sections.



Step 5: Resuming the Thread

Everything has been written out to memory space from disk and we can now proceed to resume the thread using the ResumeThread function. Our malware (or whatever you want) is now running as a process in memory! Excellent! Our crypter's job is done.

Step 6: ???

???

Step 7: Profit

Profit!

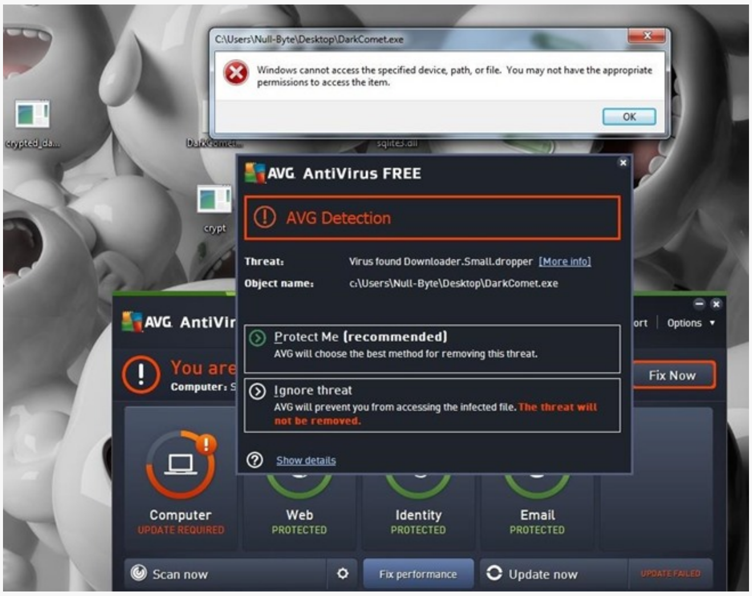
Comparing Scantime and Runtime Crypters

Before writing this article, I had tested and compared my scantime crypter and my runtime crypter. If you have not read my article on making a basic scantime crypter in C, please follow this link.

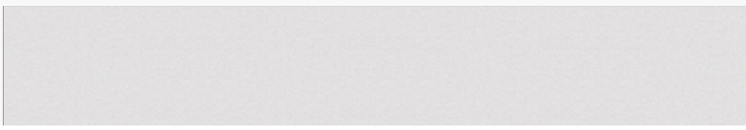
The following details into the application of these two crypters with a commercial RAT, *Dark Comet*. The antivirus used was *AVG* which is a common product for the standard user. We will be assuming that the system belongs to a standard user so we can measure the effectiveness each crypter.

Here are the files on the Desktop.

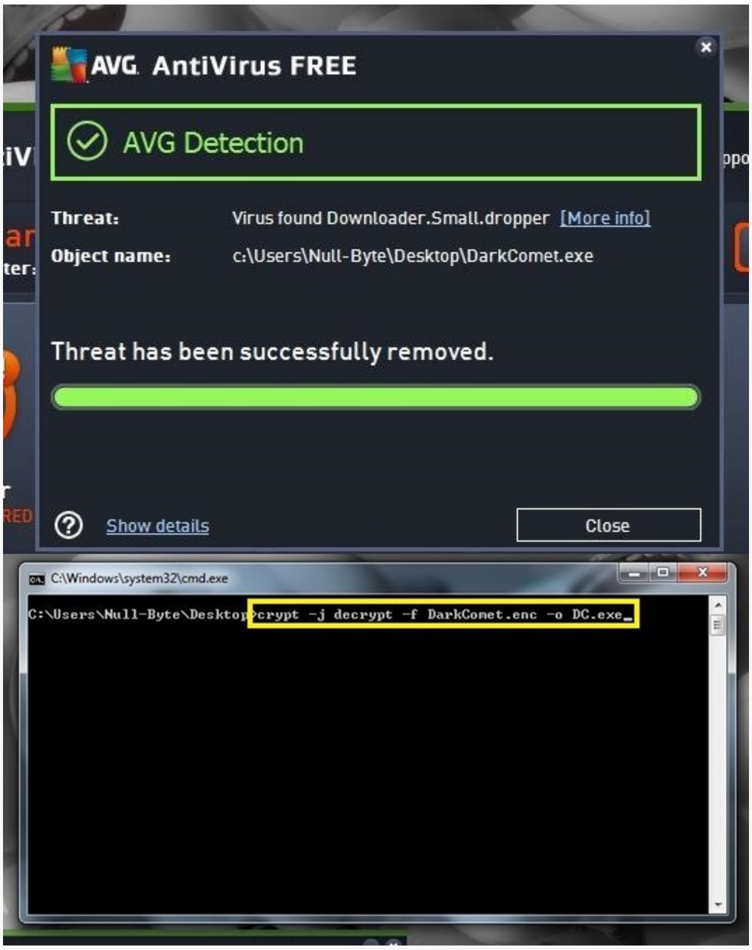
On the far left, is the runtime crypter with the obfuscated (XOR-encrypted) *Dark Comet* inside the program. In the middle is the basic scantime crypter with the obfuscated *Dark Comet* above it. On the right is the bare *Dark Comet* executable and its sqlite3.dll which is required for it to run. AVG is currently disabled. Let's enable it and see what happens.



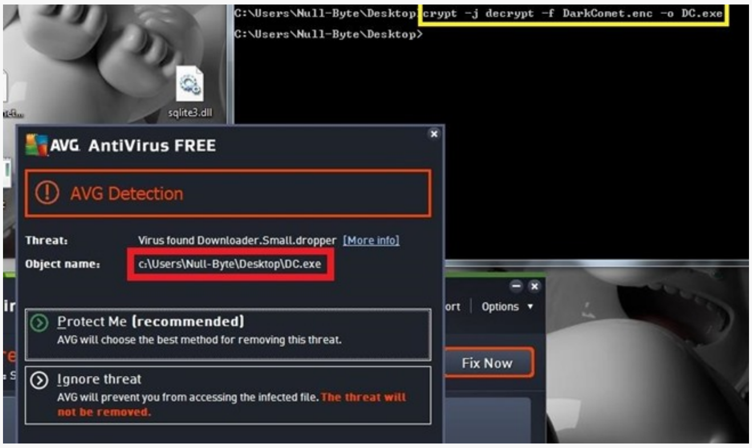
Immediately, we can see that the bare *Dark Comet* has been detected and cleansed so it is unable to be executed. I then proceeded to select the "Protect Me" option to have it removed.



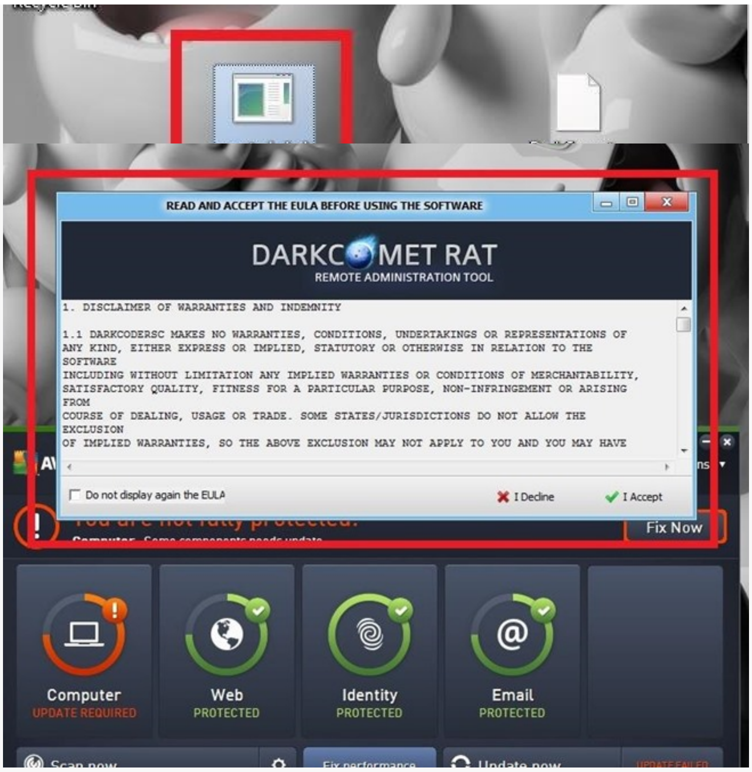
8
KUDOS



Using the command line, I attempt to decrypt the obfuscated *Dark Comet* into the *DC.exe* binary...



But it gets denied. Because it writes the file back onto disk, AVG ultimately detects it as it results in just another bare *Dark Comet* executable. No luck here! We have one more attempt with the runtime crypter.



8
KUDOS

It worked! A brilliant success! Amazing! Since it directly loads our executable in memory, it avoids the consequences of our scantime crypter and is therefore a much more effective technique. In fact, the scantime crypter had been detected by AVG so I had to add it to the exceptions list to be able to show this to you guys and gals.

Conclusion

That's it! If you feel confused, please, do some research!

Hope you've enjoyed this (somewhat lengthy, hopefully not too boring) article and now at least sort of understand what's happening! Thanks for reading!

dtm.

See Also

- Security-Oriented C Tutorial 0xFB - A Simple Crypter
- Security-Oriented C Tutorial 0xFA - Enhancing Our Crypter
- How to Speed Up & Supercharge Your HTC One

Show More...

Join the Discussion

Subscribe OFF

MENDAX V2 — 1

Nice article, hope there's more to come. BTW can you show how to get stder and stdout on windows when executing command?

23 hours ago — Reply

DONTRUSTME — 1

Thanks.

What do you mean getting stderr and stdout? Could you provide some more detail please?

22 hours ago — Reply

MENDAX V2 — 1

well I mean the output and the errors of a windows dos command.
eg.
system("whoami") will open a cmd box with the result.
I want execute the command without the box and write the error or output into a variable.

char result = command*execution*here;

22 hours ago Reply

DONTRUSTME

1

I don't recommend this but try *popen*.

22 hours ago Reply

8

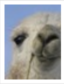
PICOFLAMINGO

1

Congratulations dtm!.

Really Great post!

23 hours ago Reply



DONTRUSTME

1

Thanks, pico!

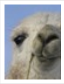
22 hours ago Reply

WINTERDRAWLACE

1

This post is utterly fantastic, as in probably the best in the entire public internet currently regarding crypters. Well done, you deserve more than kudos for your efforts

21 hours ago Reply



DONTRUSTME

1

Thanks! There are probably guides out there. They just don't have the relevant PE guide with it.

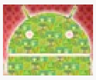
21 hours ago Reply

Share Your Thoughts

Popular How-To Topics in Computers & Programming

Hack in to another computer th...
Crack facebook password
How to Hack skype password
Hack another computer on you...
Hack router password
Hack another computer from yo..
How to Hack wifi passwords
Hack facebook account
Track who views your facebook ...
How to Hack wifi with ps3
Hack a website password
Bypass facebook password
Write in bold text on facebook w..
Trace someone else ip address
Hack facebook account password
Hack computer through wireless
How to Hack password cmd
How to Crack wifi codes
Hack other computer with ubun...
Hack wifi using command prom...
Hack another computer from yo..
Hack computer webcam
Hack password protected wifi
Hack home security camera

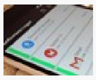
Trending Across WonderHowTo



7 Android-Only Apps That Will Make iPhone Users Green with Envy

Social Engineering - Total Guide

The 15 Best Root Apps for Android



How to Save Your Entire Notifications History on Android Forever

4 Ways to Crack a Facebook Password and How to Protect Yourself from Them

Spin Your Potatoes for Better Hash Browns at Home

Why YOU Should Study Digital Forensics

How to Perform a Cold Boot Attack?

Arts

Arts & Crafts
Beauty & Style
Dance
Fine Art
Music & Instruments

Science & Tech

Autos, Motorcycles & Planes
Computers & Programming
Disaster Preparation
Education
Electronics
Film & Theater
Software
Weapons

Lifestyle

Alcohol
Business & Money
Dating & Relationships
Diet & Health
Family
Fitness
Food
Home & Garden
Hosting & Entertaining
Language
Motivation & Self Help

Gaming

Gambling
Games
Hobbies & Toys
Magic & Parlor Tricks
Video Games