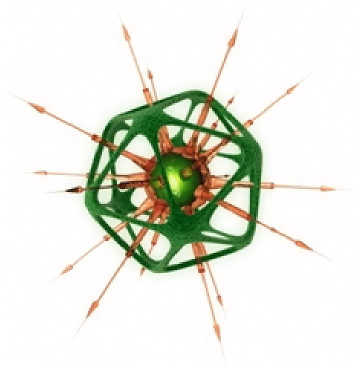
Introducing Stealth Malware Taxonomy

Joanna Rutkowska

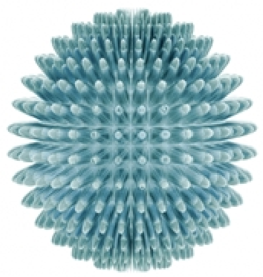
COSEINC Advanced Malware Labs

November 2006

Version 1.01


Code Infectus Stupidos

Data Infectus Smartus


Hypervisorus Blue Pillus

Introduction

At the beginning of this year, at Black Hat Federal Conference, I proposed a simple taxonomy that could be used to classify stealth malware according to how it interacts with the operating system. Since that time I have often referred to this classification as I think it is very useful in designing system integrity verification tools and talking about malware in general. Now I decided to explain this classification a bit more as well as extend it of a new type of malware - the type III malware.

Malware Definition

Before I start describing various types of malware, I would like to first define what I understand by the term *malware*:

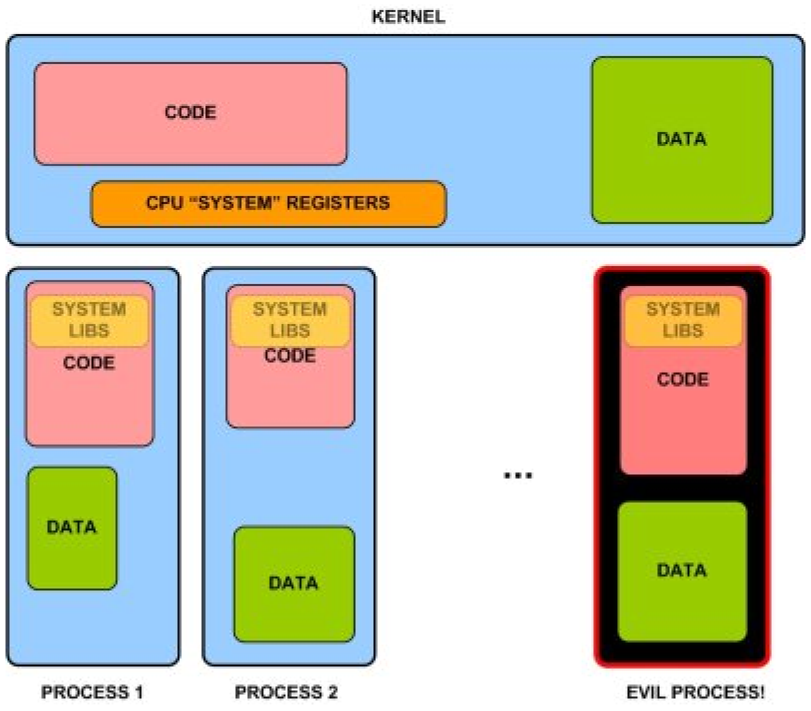
Malware is a piece of code which changes the behavior of either the operating system kernel or some security sensitive applications, without a user consent and in such a way that it is then impossible to detect those changes using a documented features of the operating system or the application (e.g. API).

The above definition is actually different from the definition used by A/V industry (read most other people), as e.g. the simple botnet agent, coded as a standalone application, which does not *hook* OS kernel nor any other application, but just listens for commands on a legally opened (i.e. opened using documented API functions) TCP port, would not be classified as *malware* by the above definition. However, for completeness, I decided to also include such programs in my taxonomy and classify them as **type 0** malware.

Below I describe each of the four classes of malware – type 0, type I, type II and finally type III and comment on the detection approach needed for each of these classes.

Type 0 Malware

As it can be seen in the picture below the malware of type 0, which, as we just agreed, is not to be considered as a *malware* from the system compromise detection point of view, does not interact with any part of the operating system (nor other processes) using any undocumented methods.



Of course, such an application (process) still could be *malicious*, e.g. it could delete all the personal files from the user's directory, or open a TCP port and become part of the botnet, possibly taking part in a DDoS attack (but again using a valid API to establish connections to the victim machines), etc.

However, looking from the *system compromise detection* point of view, all of the above behaviors are just *features* of the application and do not *compromise* the operating system nor they change (compromise) the behavior of other applications (processes) running in the system.

The A/V industry has developed lots of mechanisms to determine whether a given executable is "bad" or "good", such as behavior monitoring, sandboxing, emulation, AI-based heuristics and not to mention all the signature based approaches. Some would like to say that this is all to protect users against their own "stupidity", but it's not that simple, of course. After all, even if we assumed that we can trust some software vendors, which is, in most cases, a reasonable assumption in my opinion, and that we are smart enough to know which vendors to trust, still we download most of the applications from the internet over plain HTTP and not over HTTPS.

My favorite example is Firefox, whose binaries are available only via HTTP. Interestingly when Firefox downloads updates, it uses a secure HTTPS connection to obtain a hash value of the new binary and uses it for verification of that new update before it gets installed. However, we can never be sure that our original Firefox binary has not been compromised (as we had to download it over unsecured HTTP) so the fact the updates are "signed" doesn't help much...

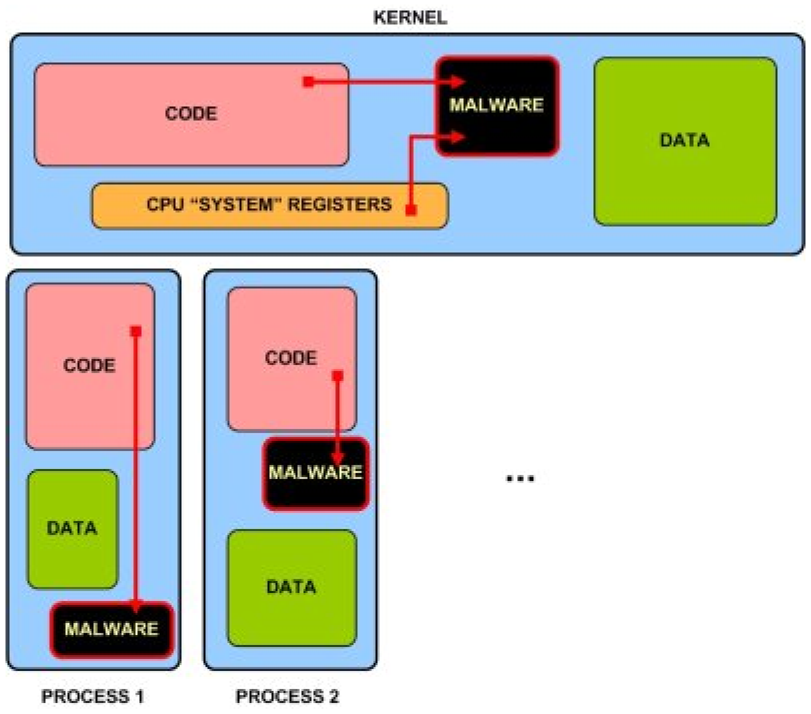
So, detecting type 0 malware is undoubtedly an important thing, especially for Jane Smith and her family, but as it is not related to system compromise detection, thus I'm ignoring this problem in my research and leave it to the A/V industry.

Type I Malware

When we look at various operating system resources, we can divide them to those which are (or at least should be) relatively constant ("read-only") and to those which are changing all the time. The examples of the former include e.g.: executable files, in-memory code sections (inside running processes and in kernel), BIOS code, PCI devices expansion EEPROMs, etc… The examples of the latter are e.g. some configuration files, some registry keys, but most importantly data sections of running processes and kernel.

The malware which modifies those resources which were designed to be constant, like e.g. in-memory code sections of the running kernel and/or processes, is something which I classify as **type I malware**. Consequently, malware which does not modify any of those constant resources, but only the resources which are dynamic by nature, like e.g. data sections, is to be classified as **type II malware**.

On the picture below an exemplary infection with type I malware has been presented:



It should be clear by now, for anybody familiar with assembler language, that there are virtually infinite ways to create type I malware of any given kind. E.g. if we consider creation of a key stroke logger, then there will be incredibly many ways of doing that by modifying (*hooking*) code at many different levels (starting from keyboard interrupt handler's code and ending at some high level functions inside applications) and in many different ways (from simple JMPs to complicated, obfuscated, execution transfers or even "code integration on place")…

So, it should also be clear that approaching type I malware detection using any kind of "find the bad" approach, like e.g. scanning for known patterns of code subversions, is an insufficient solution and is prone to the endless arm-race.

The detection of type I malware should be based, in my opinion, on verifying *integrity* of all those constant resources. In other words, on verifying that the given resource, like e.g. a code section in memory, has not been modified in *any* way. That, of course, implies that we need some *baseline* to compare with and fortunately in many case we have such a baseline. E.g. all Windows system executable files (EXE, DLL, SYS, etc.) are digitally signed. This allows us not only to verify file system integrity, but also to verify that all in-memory code sections of all system processes and kernel are intact! So, this allows us to find *any* kind of code hooking, no matter how sophisticated the hooking and obfuscating techniques have been used. This is, in fact, how my System Virginity Verifier (SVV) works [1].

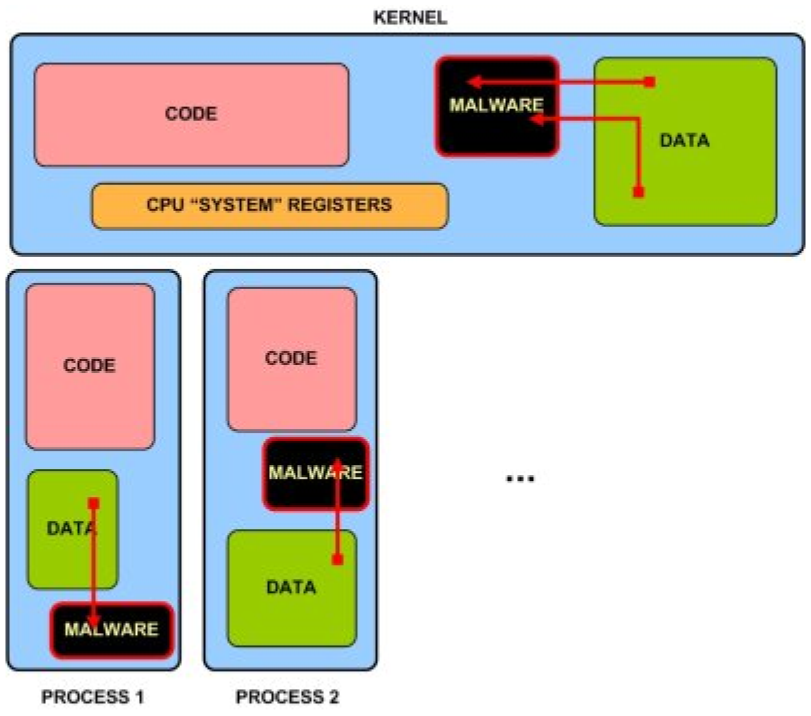
However, life is not that beautiful, and we sometimes see legal programs introducing modifications into e.g. code sections of kernel. Examples of such applications include e.g. some Host IPS products and some personal firewalls (see e.g. my BH Federal presentation for more details [2]). That disallows us to design a proper system integrity verification tool, because such a tool sometimes is not able to distinguish between a malware-like hooking and e.g. a HIPS-like-hooking, as sometimes virtually the same techniques are used by A/V vendors as by the malware authors! Needles to say this is very wrong! Probably the best way to solve this problem is the Patch Guard technology introduced in 64-bit versions of Windows. I wrote about it recently [3].

Also, there are lots of applications which are not digitally signed, so we basically can never know whether their code has been altered or not. Thus, I think that it's crucial to convince more application developers (at least the developers of the security-sensitive applications) to sign their executables with digital certificates.

Examples of the type I malware: Hacker Defender, Apropos, Sony Rootkit, Adore for Linux, Suckit for Linux, etc...

Type II Malware

In contrast to type I, malware of **type II** does not change any of the constant resources, like e.g. code sections. Type II malware operates only on dynamic resources, like data sections, e.g. by modifying some function pointers in some kernel data structures, so that the attacker's code gets executed instead of the original system or application code.



The whole point here is that the resources modified by type II malware are supposed to be modified anyway (by the operating system or the application itself), so it's very difficult to spot yet another change, this time introduced by malware. This implies that it is not possible to *automatically* verify integrity of the whole data section using a hash or a digital signature. Data sections simply can not be signed or "hashed".

In order to detect type II malware generically, we would have to analyze all data sections, belonging to not only the kernel and all its drivers but also to all the security sensitive applications running in the system. And for each such data section, we would have to identify all the instances of potentially security sensitive data structures inside this very section and for each such data structure we would have to check whether this particular data has been compromised or not.

So, we need to create a list of all the sensitive data (those *dynamic hooking places*) structures inside kernel (and potentially also inside other security sensitive applications) together with methods for their verification.

Unfortunately, creation of such a list would be undoubtedly a very hard and time consuming task. Of course, it would be somewhat easier in case we had access to the operating system or application sources, but still it seems to be a very difficult goal to achieve, especially for a 3rd party companies.

So, maybe it would be a much better idea, if the operating system vendors themselves, when designing and writing the system, also mark all those sensitive data structures in some way, so that it would be then possible to automatically verify their integrity. The same could apply to some security critical applications.

The potential argument, that such a move would only encourage malware authors to hook those dynamic resources which were published on such a list, is ridiculous. After all, we want to create such a list to be able to perform integrity checks on those dynamic hooking places, so placing a malware hook there would definitely be a bad idea for the attacker. Needles to say, attackers will find those dynamic hooking places regardless whether we will publish them or not, as it has been demonstrated already many times in the past few years. Without creating such a list, we will never be able to verify the integrity of the system.

Hopefully now it's clear why type II malware is so much more dangerous and challenging then type I malware and why it deserves its own special category in the taxonomy. It should also be clear by now, that there is no much sense to develop type II malware detection, without first solving the problem of type I malware in a generic way.

Today a complete verification of the running system is not possible, just because systems are not designed to be 'verifiable'. We rely on protection technologies only and we have no mature mechanisms to check for system compromise. This should be changed or we would never be able to trust our computers.

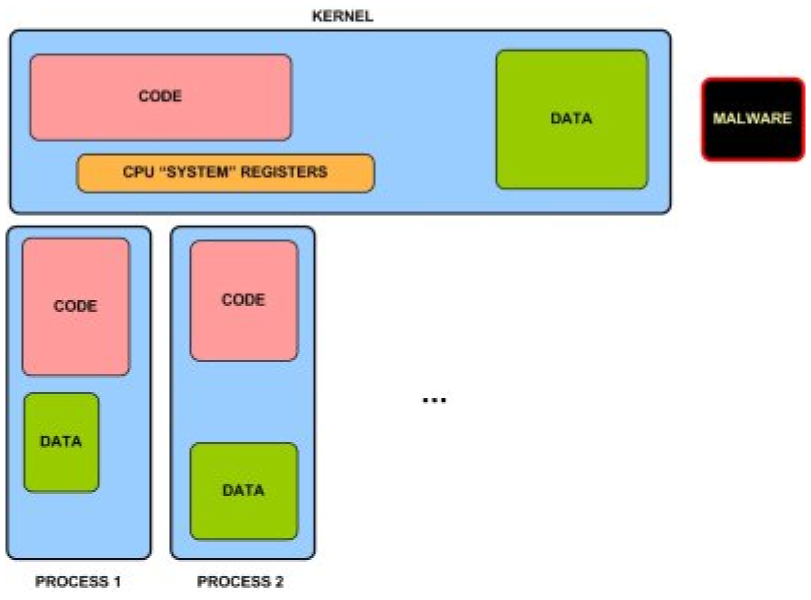
And, just to make it clear, TPM is not an easy solution to the problem. TPM is, mostly, just another prevention mechanism, although might also be useful in fighting type I malware (see above). By no means, however, TPM itself could help in type II malware detection. The same applies to the virtualization technology - having a hypervisor built into an OS, again, doesn't solve the problem of type II malware.

Examples of type II malware: deepdoor [2], firewalk [4], prrf [5], FU [6], FUTo [7] and PHIDE2 [8].

Type III Malware

Now imagine that we somehow solved the problem of type II malware (which would imply that we also solved the problem of type I malware). As I said above, this definitely is not going to be easy to achieve, but let's assume that we just created (with the help of the OS vendor) a *complete* integrity verification tool for our favorite OS. No single hooking would be able to remain undetected...

Unfortunately, this does not mean the battle is over, as apparently it has been proved recently that it's possible to create a malware which could take the control of the whole operating system, without changing a single byte in the system's memory nor software visible hardware registers! This is what I call a **type III** malware.



When I originally presented this taxonomy in January, I didn't know that it was possible to create such a malware at all, so I finished my classification with type II malware. But then, a few months later, I started researching the new hardware virtualization technology from AMD and shortly after I created a type III malware proof of concept, codenamed Blue Pill [9].

By definition, such malware can not be detected with any form of integrity scanning - thus it's virtually "100% undetectable"! And even though the current implementations of hardware virtualization on both AMD and Intel processors do not allow to effectively hide the rootkit's code, it doesn't really matter for the generic detection. What is important here is that there are no *hooks* in the system leading to the rootkit's code. So, the rootkit's code is completely "disconnected" from the system code. So, it may reside somewhere in memory, looking like some random data and there is no way for the integrity scanning tool to find out that it's the actual hypervisor code.

Of course, it doesn't mean that type III malware can not be detected by looking at some side effects it might introduce to the system. For example, one may try to perform a timing analysis of some of the instructions which are suspected of being intercepted by the malicious hypervisor. Another approach is to try to detect suspicious network activity. Unfortunately those side effects can be pretty well hidden and it might be impossible to use them in practice to detect type III malware. Also, it might be possible to detect the presence of a hypervisor by exploiting a "bug" in the virtualization implementation itself, but this again, is not a systematic way to verify system integrity, but rather a temporarily "hack".

Personally, I think, that in case of hardware virtualization based type III malware (and I don't know about any other form of type III malware), we really need to rely on prevention, as the detection is not feasible in practice here. And prevention against such malware is to have a "good" hypervisor, preferably built into the OS, which would stop the malicious ones from loading. Unfortunately creation of such protective hypervisor is a very challenging task (much more then just creation of type III malware) and it is beyond the scope of this article.

Just to make it clear - even if we implemented such a protective hypervisor, still this doesn't mean that we solved a problem of type II malware. Type II malware requires, as it has been explained in the previous section, a verification based approach, as the protection of kernel mode code will never be satisfactory in my opinion, in contrast to what we might expect from the protection of the hypervisor code.

Besides Blue Pill I'm also aware of yet another exemplary implementation of type III malware, namely the Vitriol rootkit [10], implemented by Dino Dai Zovi, which abuses Intel VT-x technology and runs on MacOS.

Conclusion

We started with type 0 malware, which is not really malware, as it doesn't introduce any changes to the operating system nor to the running applications. Then we described type I malware, which modifies things that should never be modified, like e.g. code, making it relatively easy to detect. Then we moved on to type II malware, which represents a completely different quality in malware creation and presents a big challenge for detection (integrity verification). Finally we introduced type III malware, which, at first sight, seems similar to type 0 malware, as it doesn't modify system nor applications memory in any way, but in fact is much different, as it allows to take full control of the running system and interfere with it. The current examples of type III malware uses hardware virtualization technology, but we might imagine, that in the future, some other technology will be introduced which would also allow for creation of another kind of type III malware. However, it's hard to think about something more stealthy then type III malware, thus I believe that this category closes my little malware taxonomy and makes it complete.

Credits

Thanks to Elad Efrat, who reviewed the paper and helped to improve its language quality and readability.

References

[1] Joanna Rutkowska, *System Virginity Verifier - Defining the Roadmap for Malware Detection on Windows System*, Hack In The Box 2005, http://invisiblethings.org/papers/hitb05_virginity_verifier.ppt

[2] Joanna Rutkowska, *Rootkit Hunting vs. Compromise Detection*, Black Hat Federal 2006, http://invisiblethings.org/papers/rutkowska_bhfederal2006.ppt

[3] Joanna Rutkowska, *Vista RC2 vs. pagefile attack (and some thoughts about Patch Guard)*, Invisiblethings Blog, http://theinvisiblethings.blogspot.com/2006/10/vista-rc2-vs-pagefile-attack-and- some.html

[4] Alexander Tereshkin, *Rootkits: Attacking Personal Firewalls*, Black Hat USA 2006, http://blackhat.com/presentations/bh-usa-06/BH-US-06-Tereshkin.pdf

[5] palmers, *Sub proc_root Quando Sumus (Advances in Kernel Hacking)*, Phrack Magazine, http://phrack.org/archives/58/p58-0x06

[6] FU Project, rootkit.com online magazine, http://rootkit.com/project.php?id=12

[7] Peter Silberman & C.H.A.O.S., *FUTo*, Uninformed Journal, http://uninformed.org/?v=3&a=7&t=sumry

[8] 90210, *Bypassing Klister 0.4 With No Hooks or Running a Controlled Thread Scheduler*, 29a Magazine, http://vx.netlux.org/29a/magazines/29a-8.rar

[9] Joanna Rutkowska, *Subverting Vista Kernel For Fun And Profit*, Black Hat USA 2006, http://invisiblethings.org/papers/joanna%20rutkowska%20-%20subverting%20vista%20kernel.ppt

[10] Dino Dai Zovi, *Hardware Virtualization Based Rootkits*, Black Hat USA 2006, http://blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf