Towards An Ontology of Malware Classes

Morton Swimmer*
John Jay College of Criminal Justice
445 W 59th St
New York, NY 10019

January 27, 2008

Abstract

We present a formal ontology of Malware that intends to facilitate precise communication of Malware type. The ontology consists of two parts. The first is a hierarchy of Malware characteristics that are attributed to Malware (although not necessarily exclusively so). Using these, we then create a classification of basic Malware classes by defining properties that reference the characteristics and differentiate between the classes. The ontology is expressed in OWL and publicly published so that it can be used universally in alerts, Malware descriptions and intrusion detection response systems amongst others.

1 Introduction

One of the frustrations facing administrators is the lack of accuracy in alerts from security sensors, and anti-viruses are no exception. While the fundamental problem of modeling alert messages will remain difficult, at least the field of the Semantic Web [BL98] has recently yielded some useful tools that we can use. We[1] have focussed on creating an ontology specifically of Malware classes, which is already a very broad field, because there is a great deal of confusion surrounding the common terms.

The advantage of a formal classification of Malware is that these can be referenced using URIs. If alerts are of the IETF IDMEF type [DHB04] ideally the URI representing an individual of the class should be stored in the name field of the ToolAlert. However, some properties are missing from this version that would define individuals better. Therefore the best place for this type of information would be in the AdditionalData field of the Alert class. Of course, any other type of alert format can also benefit from URI form for referencing classes.

We base our concept of Malware on *dysfunctional* software as defined in [Bru99]. If the software is intentionally dysfunctional we call it *malicious software*, *Rogue Programs* or *Malware* [Pfl97]. We can define Malware as:

Def. 1 *Malware is defined as software exhibiting unanticipated or undesired effects, caused by an agent intent on damage [Pfl97].* ■

*mailto:swimmer@acm.org

[1]This work was originally from the author's thesis, but has been based on discussions with Dr. Klaus Brunnstein and my fellow students at the University of Hamburg as well as 15 years of activity in this field. Dr. Kurt Bauknecht, University of Zürich, was instrumental in getting the process kicked off.

The agent is the programmer or it could be the person responsible for distributing a program and misrepresenting what it really does. Bugs are intentionally left out of the definition.

Intent is a difficult concept for us, because ideally, we'd like to be able to measure it, define it, or at least circumscribe it to give us better leverage on the problem of Malware detection. However, intent can only be measured at the source: we must be able to interrogate the agent (perhaps using a lie detector). Without that benefit, intent is impossible to determine. Non-withstanding, we are able to create characteristics that are useful.

1.1 Prior work

On a historic note, the effort to classify and describe Malware has been attempted many times in the past with varying degrees of success. Probably the first detailed attempt, called the *Computer Virus Catalog* [Bru90], was by the Virus Test Center at the University of Hamburg. This had been preceded by very brief and informal write-ups in lists colorfully named the "Dirty Dozen" or the "Filthy Fifty". (The naming of the list by its length became increasingly pointless as the number of Malware grew.) The Virus Catalog defined various properties that must be included in the description of a Malware individual, but did not try to classify more than viruses.

Nowadays, anti-virus vendor websites use essentially the same scheme as the Virus Catalog, but some of the terminology and methodology has evolved. In particular, Malware naming is evolving into a classification (a taxonomy to be exact) that includes a few of the basic characteristics of the Malware [SBS91, Fit03].

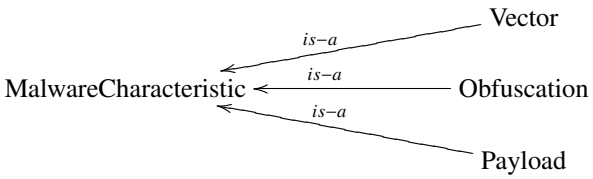
Synopsis

The rest of the paper is structured as follows: In Section 2 we will see a hierarchy of Malware characteristic classes. These are later used together with the Malware properties of Section 3.1 to form the Malware classes in Section 3.2. We conclude in Section 4 with a look at the status of the ontology and how it will be used. We are using Protégé [GMF+02] as our authoring tool to produce the OWL documents [MH04], and so we borrow some of the methodology and terminology from that tool.

2 Characteristics hierarchy of Malware

We wish to have a hierarchy of characteristic classes that we later can use in the ontology of Malware classes.

When we discuss Malware we are mainly interested in three primary categories of characteristics: its *Vector*, *Obfuscation* and *Payload*.



The Vector of a class refers to how it is deployed and spread and is the most important category of characteristics because it is often the most defining of all. Obfuscation

refers to how the Malware avoids detection. Finally, the Payload of a class is what detrimental action it performs. Each of these top-level characteristics are now split into more specific characteristics that we shall use later in the ontology of Malware classes.

2.1 Malware Vector characteristics

The Malware vector is the most important defining characteristic of a given type of Malware. It defines how the Malware is deployed or spread. There are five orthogonal sub-characteristics of Malware vectors: *plurality*, *transitivity*, *insituacy*, *locality* and *invocation*.

2.1.1 Plurality of environments

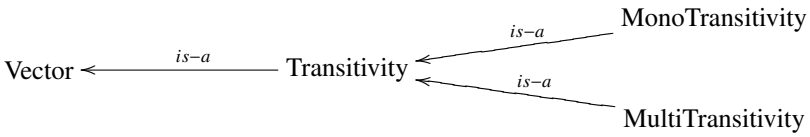
Malware that is capable of running on multiple environments have the Pluralitycharacteristic.

$$\text{Vector} \xleftarrow{is-a} \text{Plurality}$$

Actually specifying the platform(s) that the Malware can run on, is out of the scope of this paper and is an ontology in its own right.

2.1.2 Transitivity

Transitivity describes how the Malware spreads itself (if at all). Most Malware must reach its intended victim to be effective, but the question raised here is to what extent it spreads at the Malware's own initiative.



We differentiate between one hop (*MonoTransitivity*, as in the case of a dropper), or potentially infinite replication (*MultiTransitivity*, in the case of a virus).

In the MonoTransitivity case, Malware can affect a target resource by dropping a part of itself into that location. The one thing that is not copied is the copying mechanism itself and therefore it cannot spread further. For instance *trojan droppers* may modify a program to make it a trojan, or plant the trojan in a more strategic place. An exploit tool that launches a Buffer Overflow attack falls in this category as it plants the exploit code into the location where it will corrupt the stack and be run.

With *MultiTransitivity* , the Malware can potentially spread infinitely. In lab tests, a spread to three generations is the benchmark for the viability of multi-hop Malware. Of course, this type of transitivity is the defining characteristic of viruses and worms. It is a particularly useful characteristic in that the spread is directly observable, giving us a good handle on virus identification.[2]

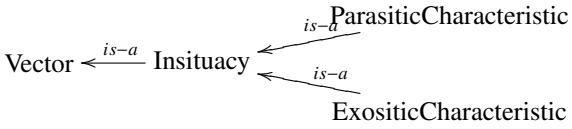
It is not necessary for the Malware to exactly maintain its form in transit and we shall see later in Section 2.2.3 that it is common to mutate in transit. In principle, it is important that it maintains the multi-transitivity characteristic through all hops. In

[2]Which is not the same thing as saying that it is easy to detect viruses, as we shall see when we look at the Obfuscation characteristics.

practice, we tolerate some corruption in the replication mechanism so long it results in a useful number of tenable replicants. If too many replicants are non-functioning, we usually classify the Malware as *intended*.[3]

2.1.3 Insituacy

Malware very often invade existing objects (usually executable ones) or profit from the existence of one in other ways. For some types of Malware, the ability to feed off an existing resource is an important part of their functionality. We define categories of insituacy[4] as *ParasiticCharacteristic* and *ExositicCharacteristic*.



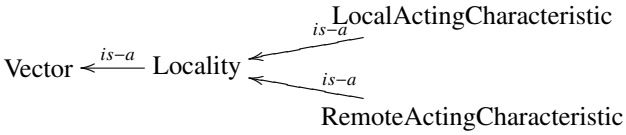
A Malware with the ParasiticCharacteristic attaches itself to its target object. The target host must be physically modified so that it includes a copy of the Malware and that it gets invoked in some manner.

The other kind of insituacy characteristic is the ExositicCharacteristic characteristic more commonly known as the *companion virus* characteristic. The target file is not actually modified, but there exists a dependency on the pre-existence of such a target file like in the ParasiticCharacteristic case. The *ExositicCharacteristic* insituacy characteristic therefore depends on the existence of an object, but the object is not modified in any way.

An example of an ExositicCharacteristic Malware is one which used a feature of the operating system called the PATH variable. This is a list of directories to search for when a program is executed. If the Malware knows the contents of this variable, it can place itself in a directory that will be searched *before* that object. As a consequence, it will be executed in its stead.

2.1.4 Locality

Transitive Malware may act locally, on the same machine, or remotely, over a network. This has consequences in which services a Malware individual is able to use, as often the network interface has less functionality than the internal local interfaces.



Some Malware are naturally local, e.g., the classical Trojan horse, and some are naturally remote acting, such as worms. Exploits usually act remotely, but some are written for local use, e.g., to elevate privileges.

When Malware acts locally, its definitive actions do not extend outside the machine it resides on, but we don't consider the payload in this. All non-transitive Malware

[3]Which is outside the scope of the current Malware ontology.

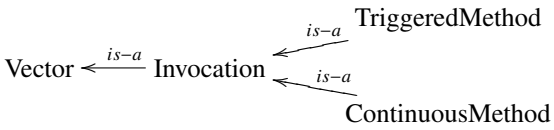
[4]The word 'in-situ-acy', derived from the Latin 'situs' meaning site, was created for this category and is meant to mean the state the Malware strives to be in through its actions. Thus, when a virus prepends itself to a file, we say its insituacy is parasitic-infix.

is also local by default as no transition can occur that would move the Malware, or part of it, off the machine. However, transitive Malware may still only operate locally. Unfortunately, this can't be adequately expressed in the OWL language we use.

If a transitive Malware acts outside of the machine it resides on, and it is not the payload that is acting, we call this remote locality. With the demise of multi-user machines, remote Malware is becoming very popular. This is evident in the rise of worms and remote exploits. Non-transitive Malware must rely on a user willfully spreading or planting it.

2.1.5 Invocation

The last of the Malware vector characteristics describes the method with which the Malware is activated, excluding payload activation. We speak of two characteristics: TriggeredMethod and ContinuousMethod.



Much like a daemon program, a Malware individual can exhibit the *Continuous-Method* characteristic when it remains in memory and sporadically acts with no obvious trigger. Inversely, it is *TriggeredMethod* if the Malware individual acts on a specific event.

The triggering can occur any number of times in the course of the Malware's existence, but only on certain predefined events. E.g., such an event can be the execution of an infected program or the receipt of a signal. The triggering mechanism may rely on the Malware being previously installed in memory or the trigger event will facilitate the loading.

A *ContinuousMethod infector* will infect at apparently random intervals not related to any particular trigger. This type is becoming very popular with worms. The attacks emitting from a worm are often truly continuous as it seeks hosts to infect by trying random IP addresses or systematically plying the network, subnet by subnet.

2.2 Obfuscation

Presumably an agent with malicious intent would find it advantageous to hide this intent, or at least make him/herself more difficult to detect. This agent can fall back on some form of camouflage making it difficult to see his/her presence. Or he can change appearance so frequently that the enemy has no easy way of detecting or tracking her/his presence. Lastly, the agent can undermine the enemies detection capability. In terms of Malware, we call this characteristic *obfuscation* and its characteristics *stealth*, *tunneling*, and *polymorphism*.

2.2.1 Stealth

We refer to the *stealth characteristic* as the ability of Malware to hide itself from an external observer. One simple form of stealth is for a Malware individual to hide the telltale ParasiticCharacteristic increase in file length from the observer using a filter as in Figure 1. If the observer attempts to retrieve the length of the infected object, the Malware, acting as a filter, will correct the length to look like the infected object's

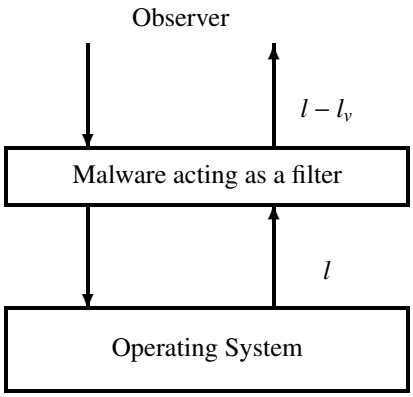
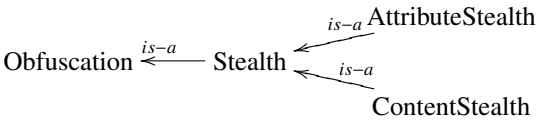


Figure 1: Malware attribute stealth characteristic: The observer reads the attributes of an individual of Malware from the OS, in this case a virus that normally has a characteristic parasitic length increase. The Malware is acting as a filter and subtracts the Malware length, l_v from the total length, l when it reports back to the observer.

original length. We call that type *AttributeStealth stealth*. A more elaborate approach is evident in *ContentStealth stealth*, where the original contents seem unaltered to the observer.



A Malware individual has *AttributeStealth stealth* characteristic if it modifies attributes of objects to avoid detection by an observer. Historically, attribute stealth was employed by some MS/PC-DOS viruses, stealth of any kind is becoming rare. In general, *ContentStealth stealth* is a more useful characteristic for Malware.

When the object is directly inspected, i.e., by reading it, the Malware hides itself, often by stripping itself from the read buffer on the fly. A Malware individual therefore exhibits the *ContentStealth stealth* characteristic if it hides its code or existence from the observer.

Content stealth was fairly popular with MS/PC-DOS viruses but hasn't seen much of a renaissance yet. Some macro viruses use the macro protection mechanisms or other methods to prevent reading of the macro virus code, which qualifies as stealth in that the observer may be tricked into believing there is no Malware. Some Windows viruses and trojans prevent their existence in memory from being seen in the process table; rootkits hide traces of their activity from log files and the like. A side effect of content stealth with parasitic Malware in files, is that sometimes a file copy operation will result in a clean copy of the file.

2.2.2 Tunneling

Despite stealth techniques, anti-Malware monitor programs usually were able to observe the Malware in action. Reacting to this, Malware was created to find the original

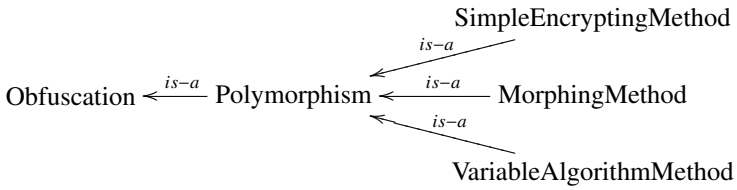
operating system handler in the chain of filters for the system call that was being monitored. The Malware could ensure that it was first in the chain just after the operating system itself by manipulating the pointers in memory. This way, the monitor was no longer able to see the Malware activity, because it happens under the monitor's level. Likewise, a behavior blocker is neither able to see the Malware nor block it. We call this the *Tunneling characteristic*.

The techniques used to hook the pointers under MS/PC-DOS would sometimes fail causing a system crash, so tunneling was difficult to execute for the not so experienced virus programmers. More modern operating system make it difficult for applications to place the required hooks in an uncontrolled manner, but it is still possible. However, there are few anti-virus monitors left on the market.

2.2.3 Polymorphism

Whereas stealth and tunneling both deal with hiding from the observer, polymorphism deals with preventing detection and positive identification, usually from scanning-based technology. The trouble that non-polymorphic Malware have, is that its manifestation is constant and once one individual has become known, knowledge-based detection systems can find them all. Polymorphism has posed a big problem to such detectors by varying the appearance of the Malware individuals.

There are varying degrees of polymorphism which correspond roughly with the amount of work required to achieve identification.[5] However, instead of proposing a quantitative scale to classify polymorphism, we will use three qualitative categories.



Entry-level polymorphism, that we call the SimpleEncryptingMethod characteristic, is encryption of a significant part of the Malware code and a *decryptor/loader* to decrypt it. In Figure 2, the decryptor/loader is shown before the encrypted Malware body and is executed first. It decrypts the body so that when execution reaches that section, the body is modified in memory to be plain-text executable code. The key for the decryption is explicitly or implicitly stored in the decryptor/loader.

Every individual of this Malware will typically generate a different key before encrypting the plain-text body and storing it to disk. The decryptor/loader is also stored to disk with the current key so that decryption can commence when the infected file is executed. Thus the Malware changes its representation on disk from instance to instance. It should be noted that the decryptor/loader is still plain-text and invariant except for the key, making it suitable for detection using string matching.

More advanced polymorphism, that we call *MorphingMethod*, involves obfuscation of the decryptor/loader and possibly the entire Malware. The methods vary, but a good starting point is code permutation.

Virus writers have created quite a variety of tricks for obfuscating the decryptor loader and making detection difficult. They further enhanced their technology by using

[5]For detection, if not identification, is still possible using various techniques outside the scope of this paper.

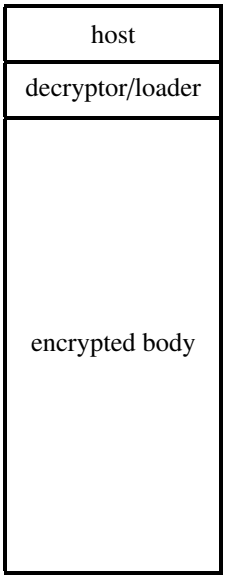


Figure 2: Structure of a simple encrypted Malware

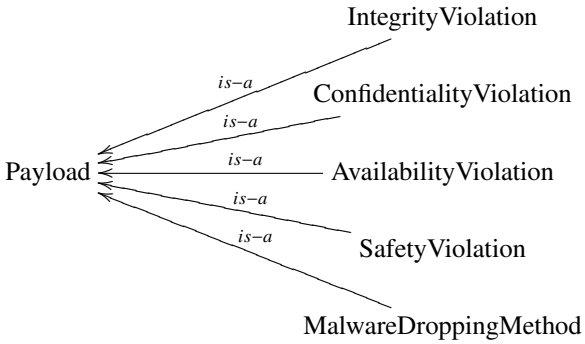
anti-debug and anti-emulator tricks, that prevent easy analysis in a debugger or execution in a CPU emulator. It is outside the scope of this paper to describe these techniques in detail.

The last polymorphic method goes in a different direction. Instead of being fixed, the algorithm itself is one of a given set. We call this *VariableAlgorithmMethod* polymorphism. Different mathematical functions can be used, including quite sophisticated algorithms, although given that in nearly all cases, the Malware must carry the key together in its code, a sophisticated algorithm only buys a time-delay when decrypting it.[6]

2.3 Payload

The payload is often of paramount importance to the affected user. As in the study of classical computer security, we classify the payload's capabilities as integrity, confidentiality, safety and availability violating. To this we add a "recursive" characteristic: the Malware dropping characteristic. Arguably, all viruses have at least implicitly a payload, even if this is merely the infection mechanism of a virus violating a system's integrity or availability. We also view the question of intent as important: the alleged payload must be programmed with malicious intent, as far as we can judge.

[6]This is something that may be significant when using generic decryptors that anti-virus products commonly use. These are based on emulating the decryptor/loader code, which is a slow process to begin with. However, such techniques are rarely a hindrance except when environmental keys [RS98] are used.



2.3.1 Integrity violating

Violating the integrity of system resources is probably the most effective way of wreaking havoc on a system. If done subtly, the damage can be extensive before it is noticed, by which time backups will contain corrupted data. There have been examples of this in the DOS/DarkAvenger and the W97/Wazzu viruses.

Of course, parasitic self-replicating Malware are integrity violating by their very nature. They modify executable files and therefore violate the file's integrity and that of the system in many cases. Some transitive Malware is also confidentiality violating. As they spread, they copy all or parts of the documents they spread from. Some MS--Word worms are examples of this. However, we typically don't call these Payloads.

2.3.2 Confidentiality violating

Far more problematic is Malware that breach the defenses of a computer and leak data to a third party. An interesting example of this (and a good example of a clueless agent [RS98]) is the DOS/Cheeba virus [Gry92] that stole the password files from BBS systems (and did it in a way that made analysis of the payload extremely difficult).

Trojan horses are most often confidentiality violating, or at least a vehicle for such violation. There were very early password stealing trojans from the time of mainframes and terminals. More recently, trojans have been used as back doors into systems such as with W32/Bo2k.

2.3.3 Availability violating

Availability is violated when a resource becomes inaccessible to a legitimate user. This is commonly known as a *Denial of Service* attack (*DoS*). An example of this is the *Ping of Death* that affected many systems by sending a malformed[7] ICMP packet to the target machined. A particularly nasty variation of this is the case of a *DDoS* attack (*Distributed Denial of Service*). These can (but don't have to be) facilitated by viruses or worms.

The availability violation may also be more permanent. The DOS/AntiCMOS virus attempted (but failed due to a bug) to erase the CMOS RAM in PCs, which would have made for a problematic recovery. Modern motherboards also store the BIOS in EEPROM[8] memory leading to speculations of viruses that overwrite the BIOS.

[7] The packet length typically exceeded the allowed size.

[8] Electrically Erasable Programmable Read Only Memory

2.3.4 Safety violating

Safety is violated when a manipulation in the processes of a system intensionally results in an endangerment of a (human) being. It has only been recently recognized as a separate (albeit related) discipline to the big C, I and A of computer security. Safety-critical systems are those where a failure would likely lead to the loss of lives or the bodily harm at the very least.

Malware can interact maliciously with such systems in a way that will cause such bodily damage.

2.3.5 Malware dropping

A less direct payload is the dropping of another piece of Malware. A worm, e.g., W32/-CodeRed-II, may drop a back door trojan so that even when the system is cleaned of the actual worm, a hacker can access the affected system. A good anti-virus program should clean up both, but an ad hoc attempt may miss the trojan.

Of course, an exploit will usually drop a trojan or shellcode, but where we must draw the line is were worms use exploits to propagate. In this case, we perceive the exploit code to be a part of the worm as it is a core part of the worm's functionality.

Note that this characteristic is not the same as plurality discussed in Section 2.1.1 which can be viewed as alternate code variations for different environments.

2.4 Summary of the Malware characteristics

In Figure 3 we see the entire hierarchy of Malware characteristics. The ontology is authored in OWL [MH04] and is available at [Swi04a]. In the next section we will use these characteristics to define the ontology of Malware.

3 The ontology of Malware

Given the solid foundation of characteristics we wish to base our ontology on, we now need some basic properties before we can define some common Malware terms.

3.1 Properties

Ontology properties define aspects of the classes that allow us to differentiate between classes. Ideally, classes should distinguish themselves by having different properties or different restrictions on the properties.

Properties are defined as verbs in a phrase: the subject, the *domain*, is the class itself, and the object, the *range*, is often another class, an individual of a class, or some value. However, in defining the properties of classes, we don't always make the object assignment, but often just list the properties that apply, possibly with some restrictions. The restrictions can be a specific class or a set of classes, or merely a cardinality of the set.

3.1.1 drops

The *drops* property defines which classes of Malware this Malware drops.

Figure 3: The class hierarchy of Malware characteristics

3.1.2 hasPayload

The *hasPayload* property defines what kinds of payload is contained in the Malware. The range of this property is therefore *Payload* .

3.1.3 hasInsituacy

The *hasInsituacy* property defines which form of Insituacy is allowed. We only allow a single value.

3.1.4 hasLocality

The *hasLocality* property defines whether the Malware operates LocalActingCharacteristic or RemoteActingCharacteristic. The range is therefore the Locality.

3.1.5 hasObfuscation

The *hasObfuscation* property defines which types of Obfuscation the Malware uses.

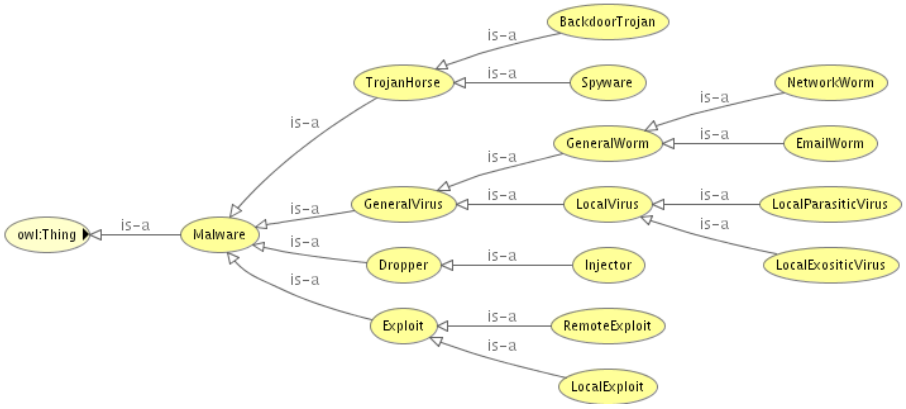


Figure 4: The Malware class hierarchy

3.1.6 hasTransitivity

The *hasTransitivity* property defines which type of Transitivity is used by the Malware (if any.) Only one value is allowed.

3.1.7 hasPlurality

The *hasPlurality* defines whether the Malware functions on multiple platforms or not. The range is therefore Plurality in the characteristic hierarchy.

3.2 Classes

In Figure 4 are all the classes of Malware that we will consider. Of course, given the characteristics and the properties we just discussed, there are far more classes of Malware possible than just this. However, in this paper we will only cover the most common types and leave a more expanded class hierarchy for later. It should also become clear that given the properties, a different tree structure could equally well have been chosen. However, this structure reflects the most common perception of class relationships, and it is the job of a good ontology to reflect the common perception of terms.

The root of this ontology is the term Malware and it contains only three properties:

$$\text{Props(Malware)} = \{\text{hasPlurality}, \text{hasObfuscation}, \text{hasPayload}\} \quad (1)$$

because these are universal properties that any Malware can have. However, no restrictions have been set for Malware.

3.2.1 Viruses in the typology

In this ontology, we define worms to be separate from viruses. As a catch-all for the entire group, we create the class of GeneralVirus for those properties that are common to both viruses and worms in this ontology.

$$\text{Props(GeneralVirus)} = \left\{ \begin{array}{l} \text{hasPlurality, hasInsituacy, hasLocality,} \\ \text{hasTransitivity} \ni \text{MultiTransitivity,} \\ \text{hasObfuscation, hasPayload} \end{array} \right\} \quad (2)$$

We have added a few properties to Props(Malware) and one of them has a restriction: hasTransitivity ∋ MultiTransitivity. We use the ∋ symbol to mean "has value"[9], meaning in this case we restrict the the hasTransitivity property to the single value MultiTransitivity.

This is a very general renderation of the term virus and in the following equation, we refine it down to what we more commonly call a virus and in this ontology we call a LocalVirus.

$$\text{Props(LocalVirus)} = \left\{ \begin{array}{l} \text{hasPlurality, hasInsituacy,} \\ \text{hasLocality} \ni \text{LocalActingCharacteristic,} \\ \text{hasTransitivity} \ni \text{MultiTransitivity,} \\ \text{hasObfuscation, hasPayload} \end{array} \right\} \quad (3)$$

Over the GeneralVirus, we have the further restriction that the locality must have the value LocalActingCharacteristic. There are two further refinements that we can make.

$$\text{Props(LocalExositicVirus)} = \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation, hasPayload} \\ \text{hasInsituacy} \ni \text{ExositicCharacteristic,} \\ \text{hasLocality} \ni \text{LocalActingCharacteristic,} \\ \text{hasTransitivity} \ni \text{MultiTransitivity} \end{array} \right\} \quad (4)$$

$$\text{Props(LocalParasiticVirus)} = \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation, hasPayload,} \\ \text{hasInsituacy} \ni \text{ParasiticCharacteristic,} \\ \text{hasLocality} \ni \text{LocalActingCharacteristic,} \\ \text{hasTransitivity} \ni \text{MultiTransitivity} \end{array} \right\} \quad (5)$$

The LocalExositicVirus adds the restriction that the virus must be exositic and the LocalParasiticVirus adds the parasitic restriction to the virus' insituacy property. As with all siblings in this ontology, ideally they are mutually exclusive. However, in practice and with the hasPlurality property, it is difficult to have both exact definitions and clean ontology.

3.2.2 Worms in the typology

$$\text{Props(GeneralWorm)} = \left\{ \begin{array}{l} \text{hasPlurality, hasInsituacy,} \\ \text{hasLocality} \ni \text{RemoteActingCharacteristic,} \\ \text{hasTransitivity} \ni \text{MultiTransitivity,} \\ \text{hasObfuscation, hasPayload} \end{array} \right\} \quad (6)$$

The term "worm" usually defined to mean "network-aware virus".[10] But this ontology shows that it is more elegant to treat them separately. The similarity to viruses comes from a worm's and virus' MultiTransitivity properties. The addition to the properties in (2) is in restricting the locality to RemoteActingCharacteristic.

[9]The terminology and symbol set is adopted from Protégé [GMF+02]

[10]Notation warning! Some literature define worms to be non-parasitic viruses, with no implicit network-awareness. While it is true that many worms infect using ExositicCharacteristic insituacy, we feel that the network-awareness is the more important differentiating property.

3.2.3 Trojans in the typology

$$\text{Props(TrojanHorse)} = \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation,} \\ \text{hasPayload} \geq 1 \end{array} \right\} \quad (7)$$

$$\text{Props(BackdoorTrojan)} = \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation,} \\ \text{hasLocality} \ni \text{RemoteActingCharacteristic,} \\ \text{hasPayload} \geq 1 \end{array} \right\} \quad (8)$$

$$\text{Props(Spyware)} = \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation,} \\ \text{hasLocality} \ni \text{RemoteActingCharacteristic,} \\ \text{hasPayload} \ni \text{ConfidentialityViolation} \end{array} \right\} \quad (9)$$

Trojan horses, in its common use, are so vaguely defined that they are nearly indistinct from Malware, except for one thing. The cardinality of the hasPayload property must be at least one, as expressed by the hasPayload ≥ 1 expression.

The refinement of BackdoorTrojan adds remote locality to (7). Spyware is even more specific in that it is confidentiality violating.

3.2.4 Exploits in the typology

$$\text{Props(Exploit)} = \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation, hasPayload} \\ \text{hasInsituacy} \ni \text{ParasiticCharacteristic,} \\ \text{hasLocality} \end{array} \right\} \quad (10)$$

$$\text{Props(LocalExploit)} = \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation, hasPayload} \\ \text{hasInsituacy} \ni \text{ParasiticCharacteristic,} \\ \text{hasLocality} \ni \text{LocalActingCharacteristic} \end{array} \right\} \quad (11)$$

$$\text{Props(RemoteExploit)} = \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation, hasPayload} \\ \text{hasInsituacy} \ni \text{ParasiticCharacteristic,} \\ \text{hasLocality} \ni \text{RemoteActingCharacteristic} \end{array} \right\} \quad (12)$$

We shall use the term *exploit* to mean those programs that inject code into a running program. The job of the exploit is to establish communication with the target program and bring it to the point where the code can be inserted. An exploit is unlike ordinary droppers (13) in that it injects code into a running program and so we restrict the hasInsituacy property.

We also define two further subclasses: the LocalExploit and the RemoteExploit that restrict the hasLocality property in (10) to LocalActingCharacteristic and RemoteActingCharacteristic respectively.

3.2.5 Droppers in the typology

$$\text{Props(Dropper)} = \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation, hasPayload} \\ \text{hasInsituacy} \ni \text{ExositicCharacteristic,} \\ \text{hasLocality} \ni \text{LocalActingCharacteristic,} \\ \text{hasTransitivity} \ni \text{MonoTransitivity,} \\ \text{drops} \exists \text{Malware,} \end{array} \right\} \quad (13)$$

$$\text{Props(Injector)} \quad = \quad \left\{ \begin{array}{l} \text{hasPlurality, hasObfuscation, hasPayload} \\ \text{hasInsituacy} \ni \text{ExositicCharacteristic,} \\ \text{hasLocality} \ni \text{LocalActingCharacteristic,} \\ \text{hasTransitivity} \ni \text{MonoTransitivity,} \\ \text{drops} \ni \text{GeneralVirus,} \end{array} \right\} \tag{14}$$

A dropper is a general concept for a type of Malware that has a Malware dropping payload and it's property restrictions are given in (13). In a sense, it is a form of TrojanHorse, but this class tends to be treated separately. An injector, as defined in [Bon98], is a specialization of the Dropper that drops a GeneralVirus and is often found in virus collections as tools that cause the initial infection.

4 Conclusions

We have seen this very basic ontology of Malware classes based on a concise set of properties and characteristics. It is not meant to be exhaustive, but the hope is that it will be good enough to kick off a discussion and solidify the existing classes so that they are widely accepted, and then add new classes.

Once accepted the use of such an ontology is manifold. For one, it should greatly reduce ambiguity when reading Malware descriptions. Currently, the texts are authored by a variety of people even within a single organization and a sampling of Malware definitions from many sources, showed that the terms were not being used consistently even within an organization!

Furthermore, a clear ontology should allow Malware descriptions to be machine readable so that their content can be reasoned on. For instance, if a Malware description XML document wishes to describe an individual of some Malware class, it can be referenced simply as `http://www.swimmer.org/2004/11/08/malware/malwareclasses#LocalParasiticVirus` meaning that the individual is a LocalParasiticVirus as defined in the 2004/11/08 version of the Malware classes ontology. Precise information from alerts can be used in intrusion detection and response tools to respond in an appropriate manner without the guesswork that is currently required. Currently, the only information such tools have at their disposal is the fact that an anti-virus has detected some type of Malware so a measured response to a LocalParasiticVirus is not possible and we must take all possible characteristics into account.

Obviously, this project is still in its very beginning. More properties and characteristics must be defined that will allow us to differentiate between more subclasses of Malware. Other annotational properties (such as name, length, ...) need to be defined to be able to declare individuals of classes properly. It is hoped that an elegant solution will be found for Malware with the hasPlurality property set to MultiEnvironment.

The OWL documents for the current version of the characteristics and the Malware classes are available at [Swi04a] and [Swi04b] respectively.

References

[BL98] Tim Berners-Lee. Semantic web road map. Internet, October 1998. URL `http://www.w3.org/DesignIssues/Semantic.html`.

[Bon98] Vesselin V. Bontchev. *Methodology of Computer Anti-Virus Research*. Ph.D. thesis, Fachbereich Informatik der Universität Hamburg, Germany, 1998.

[Bru90] Dr. Klaus Brunnstein. Zur Klassifiaktion von Computer-Viren: der Computer Virus Katalog. In *GI-Jahrestagung*, pages 498–509. 1990.

[Bru99] Dr. Klaus Brunnstein. From antivirus to antimalware software and beyond: Another approach to the protection of customers from dysfunctional system behaviour. In *22nd National Information Systems Security Conference*. 1999.

[DHB04] D. Curry, H. Debar, and B. Feinstein. The Intrusion Detection Message Exchange Format. *Technical report*, Internet Engineering Task Force, 8 July 2004. URL `http://www.ietf.org/Internet-drafts/draft-ietf-idwg-idmef-xml-11.txt`.

[Fit03] Nick Fitzgerald. A virus by any other name - virus naming updated. *Virus Bulletin*, January 2003. URL `http://www.virusbtn.com/magazine/archives/200301/caro.xml`.

[GMF+02] J. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S. W. Tu. The evolution of Protégé: An environment for knowledge-based systems development. *Technical Report SMI-2002-0943*, University of Stanford, Stanford Medical Informatics, 2002. URL `http://smi.stanford.edu/pubs/SMI_Abstracts/SMI-2002-0943.html`.

[Gry92] Dmitry Gryaznov. Analyzing the Cheeba virus. In *Proceedings of the 1992 EICAR Conference, Munich, Germany*, pages 124–136. 1992.

[MH04] Eric Miller and Jim Hendler. Web ontology language (OWL). Internet, September 2004. URL `http://www.w3.org/2004/OWL/`.

[Pfl97] Charles P. Pfleeger. *Security in Computing*. Prentice-Hall, second edition, 1997.

[RS98] James Riordan and Bruce Schneier. Environmental key generation towards clueless agents. In Giovanni Vigna (Editor) *Mobile Agents and Security*, number 1419 in Lecture Notes in Computer Science State-of-the-Art Survey, pages 15–24. Springer-Verlag, Berlin Germany, 1998.

[SBS91] Fridrik Skulason, Vesselin Bontchev, and Alan Solomon. A New Virus Naming Convention. Internet, 1991. URL `http://www.virusbtn.com/files/naming.txt`.

[Swi04a] Morton Swimmer. Malware characteristics ontology. Internet, November 2004. URL `http://www.swimmer.org/2004/11/08/malware/characteristics.owl`.

[Swi04b] Morton Swimmer. Malware classes ontology. Internet, November 2004. URL `http://www.swimmer.org/2004/11/08/malware/malwareclasses.owl`.