Computing, Information Systems, Development Informatics & Allied Research Journal
Vol. 5 No. 2. June 2014 – www.isteams.net/cisdijournal

ENHANCING SAFER COMPUTATION THROUGH A BETTER KNOWLEDGE OF VIRUS INTERNAL MECHANISMS ON COMPUTING PLATFORMS

[1]Odule, T.J. & [2]Awodele, O.
Department of Mathematical Sciences, Olabisi Onabanjo University, P.M.B. 2002 Ago-Iwoye, Nigeria
Department of Computer Science, Babcock University, Ilishan Remo, Nigeria.
E-mail: tola.odule@oouagoiwoye.edu.ng; awodeleo@babcock.edu.ng

ABSTRACT

An exposition of the nature, structure and mode of operation of computer viruses is hereby presented, with the assertion that computer viruses are best managed through a proactive measure using a cryptographically strong checksumming algorithm that generates a fingerprint for each executable stored on the system. Verification is then performed prior to running an executable file to ascertain that the system is in a 'safe state' since the last run. This ensures inoculation against all present and future viruses with absolute guarantee.

Keywords: Executable-file, cryptographic-checksum, interrupt-vector, encryption, signature.

1. INTRODUCTION

A virus is an infectious computer program that attaches itself to executable files, boot sector of a floppy disk, partition sector of a hard disk, batch files and device drivers to hide and propagate itself without immediately revealing its presence on the computer system. Viral programs have three main variants [1] namely: Trojan horses, worms and virus. The main difference among these variants is in their mode of operation. A Trojan horse, in every sense, resembles and works like a normal computer program. Each time the program calls for keyboard input and a response is made, it may result in the reprogramming of the keyboard keys such as 'd' key expanding to 'del' thus deleting a specified medium. This type of viral program is normally propagated via device drivers like ANSI.SYS using the ANSI escape sequences or the dynamic link libraries using one of several sophisticated methods. Worms, unlike a virus recreate completely, making precise duplicates of themselves. Multiuser systems and electronic communication networks are natural homes to worms. They propagate through multiple computer connections network communication channels. Worms have the effect of slowing down the processing speed of the computer, which may be disruptive in a real-time process. Viruses and worms are best defined by replication, executable path, side effects and disguise [2].

2. CLASSIFICATION OF VIRUSES

Viruses are classified into two groups in accordance with the runnable program file infected: ***Boot sector*** virus and ***parasitic*** virus

Bootstrap sector virus [3] changes the substance of either the disk ***bootstrap sector*** or the segment bootstrap area, contingent upon the infection and disc nature, ultimately supplanting the authentic substance with its own form. The first form of the changed area is ordinarily put away elsewhere within the disc, such that during system start-up, the infected copy first gets executed. The rest of the infected program is thereafter brought into memory, immediately matching it with the invocation of the first form of the ***Bootstrap sector***. The infection consequently persists in memory-till the PC is turned off.

A ***bootstrap sector*** virus typically employs three different constituent parts in its technology:

- The ***bootstrap sector*** supplanted with an infected copy, thus paving way for the virus.
- A hitherto free sector—meant to keep the initial ***bootstrap sector***.
- Several formerly free areas on disk —meant to keep majority the viral program.

Examples are ***Brain***—a removable disc ***bootstrap sector***, ***Italian*** as well as ***New Zealand*** both of which removable and fixed disc ***partition bootstrap sectors***.

modify The internal makeup of executable files including device drivers are usually modified by ***parasitic viruses*** [4]. These viruses embed themselves toward the end or beginning of the executables while the bulk of the target code is left unchanged. The first branching instruction in the program is modified, as is the case in *Vienna* and *Datacrime*, though application efficiency is typically maintained. Although, quite a number of viral programs exist which overwrites the initial program entry point, thus rendering it ineffective. While the methodology may appear to be less contagious compared to that employed by terminate-and-stay resident (TSR) viral programs, [5] the infection effects is equally damaging or even worse in contrast to the TSRs. In addition, tracing them is problematic due to the fact that the vector table and contents of accessible storage are left intact while their infective performance are naturally very irregular.

Hybrids: Both aforementioned techniques are featured by some viral codes. An example of these is ***Typo Virus*** [6], which corrupts program files when executed, leaving slight traces of terminate-and-stay resident portion in storage in the aftermath of corruption. The payload is inherent in the TSR segment whereas the self-reproducing program statements are held offline. Variants of these viral programs may be coded to function in ways quite different from these descriptions.

3.1 Manifestation of viruses

Viruses use hiding mechanisms, which allow them to replicate unnoticed, before delivering the 'payload'. Viruses employ two hiding mechanisms: *encryption* and *interrupt interception*.

3.2 Encryption

A handful of viral programs encrypt their instruction sequence such that the bulk of this sequence looks dissimilar in every program file infected. The sole intention of this is to thwart any effort aimed at dissecting any known sample--***signature***--from the viral program as it mutates once it attacks an application [7] as shown in Figure 1.

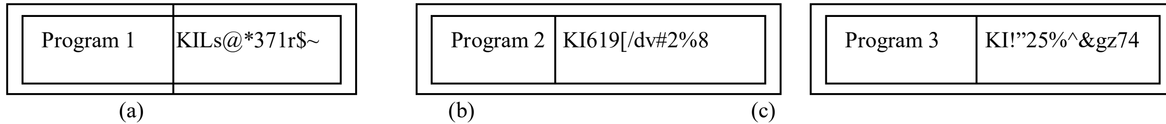


Figure 1 Three viruses infected with an identical encrypted virus

Prior to being invoked, the viral program is decoded to make the code expressive and executable. It is a requirement that the decoding module is in plain text format, ideally comprising between ten and twenty characters, uniquely applicable to all corrupted program files depicted as 'KI' in Fig. 1. The encryption key is usually linked, mathematically or otherwise, to the length of the executable.

The opportunities for presenting confusions in encrypted viral programs are essentially limitless. It is possible, for instance, to embed the key for another round of encryption in coded manner in the first round in a double round encryption. Alternatively, cryptographically stronger algorithm may be used in place of the simple functions like exclusive-or (*XOR)* as in *Cascade* [8].

3.3 Interrupt Interception

Virus presence may effectively be concealed in an infected PC by using interrupt interception. Most PC-based user programs deploy program interrupts to interface the operating system in a compact manner [9]. Branching locations are, for example, contained in the vector table placed at the start pf program memory, Fig. 2, so that upon the generation of an interrupt by an application, there is a branch to a programmed location.

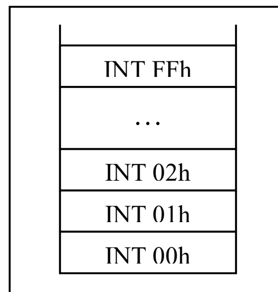
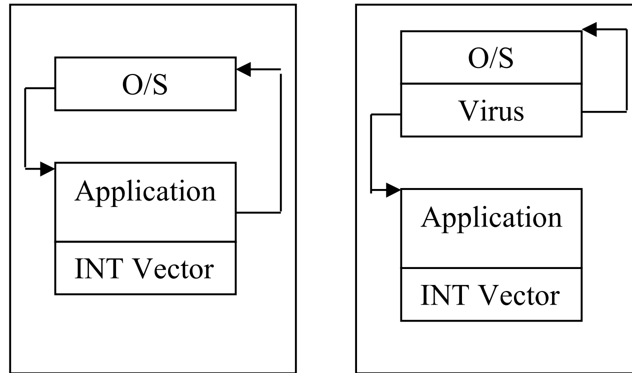


Figure 2: Interrupt Table

When a call is thus made via the operating system, it is intercepted by the viral program, which consequently delivers its payload upon the receipt of such request; since several of such locations may be changed by a virus, as shown in Figure 3. The *Brain* virus [10] uses this technique.



(a) (b)

Figure 3: Interrupt routing before and after infection

3.4 Binary Viruses

Binary viruses constitute distinctive instances of scrambled viral programs. The principle is that a viral program contains the reproducing instruction sequence in its entirety, with about 50% of its attendant effect. Just upon the invocation of the accompanying part of the infectious code conveying the other portion of the effect, the mix of the two halves results in executable expressive instructional sequence. as shown in Figure 4. This combination may be accomplished through the use of ***XOR*** procedure on both parts. However, conducting investigations on the attendant effects of a ***binary virus*** is impossible without being in possession of both parts.

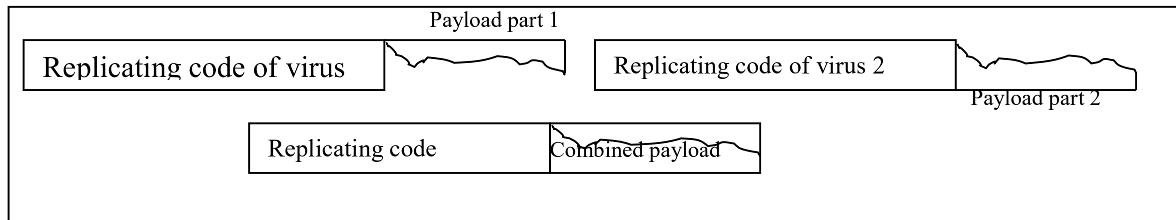


Figure 4: Binary virus—two parts combining to get a meaningful payload

While this notion was viewed as a dangerous development in virus sophistication, it has not been widespread. One of the very few viruses that seem to have incorporated this concept is the ***dBASE virus***. As a component of the attendant effect, the initial viral code includes instruction run shown in Figure 5 on Intel machine and its classes [11]:

```
        CLI
        MOV     AX,3        ; Set count
LABEL:  MOV     CX,100h
        MOV     DX,0        ; Page 0 RAM
        MOV     DS,DX       ; Segment 0
        XOR     BX,BX       ; Offset 0
        PUSH    AX          ; Save the count
        INT     3h
        INT     3h
        POP     AX          ;Restore count
        INC     AX          ; Increase count by 1
        CMP     AL,1Ah      ; Count = 26?
        JL      LABEL       ;Repeat code if less
        …
```

Figure 5: Specimen code sequence of a Binary virus

This arrangement is functionally trivial until one of the ensuing criteria holds:

1. A companion virus changes the two INT 3h instructions into one INT 26h instruction.
2. A companion virus alters the vector table such that the INT 3h references INT26h.

In the event that one of the aforementioned criteria occurs, the attendant effect turns out to be exceptionally ruinous. Upon invocation, the altered viral code will overwrite the first 256 *sectors* of drives *D*: to *Z*: making use of unconditional disc-write command (*INT 26h*).

4. RECOGNITION OF VIRUS SIGNATURES

A basic method used in checking program files for infections starts with extracting recognition strings in familiar viral programs. Such strings are typically denoted in hexadecimal notations and technically regarded as *hex patterns* or signatures. [12]. The hex patterns used to be between 10 and 16 characters in length with a very slight, limited probability of finding a part of this *hex string* in some clean and harmless program files. Data distribution in runnable program files are not totally arbitrary, making it possible to find some order of statements constituting a viral program as part of a normal, innocuous program file. It is a convention, however, to choose the signature of a viral program such that the probability of it being found in an authentic, uninfected program file is nil, though this probability may fail. If a pattern checking program reports a pattern match, it does not necessarily mean that a virus has been found, but that a virus may have been found.

4.1 Analysis of a viral Program

Upon the discovery of an infection, it is instructive to extract a signature for the purpose of examination, since analysis may be of assistance to different locales having similar infection. Regardless of whether an infection is totally dissected instantly or not, obtaining a *hex string* is important in order to assist in recognizing events of a similar infection somewhere else. A complete scrutiny of an infection naturally results in its total breakdown—analysing the construction and operation of the viral program in order to recreate workings of the executable target code as remarked source statements. Analysis of a viral program may be accomplished with DEBUG, a tool provided as a feature of MS-DOS.

4.2 Virus Disassembly

Analysing a viral program is a repetitive procedure that begins with determining what sections of the program contain data, which are to be left intact in the analysis, as well as those that contain statements. Having determined this, the results of the analysis are diverted to a sink that holds the dissected viral program. Figure 6 is an illustration of the order of DEBUG constructs contained in a document called INSTR used for the analysis of a theoretical viral program held on a sink called VIR.COM.

```
U   100   102
D   103   10F
U   110   432
Q
```

Figure 6: Specimen sequence of DEBUG commands

Invoking DEBUG via the instruction *DEBUG VIR.com < INSTR > VIR.asm*, causes *INSTR* to act as the source file from which the input is read, and *VIR.asm* as the sink holding the dissected results of the viral program, *VIR.com*. Analysing a *boot sector* viral program may involve a fairly quite difficult process since they use larger areas than simply the *boot sector*. Analysis of the boot sector should first be undertaken to determine other areas utilized by the viral program. The technique for diverting the source and sink of DEBUG may equally be employed in the analysis of *parasitic viruses*.

Suppose a viral program occupies sectors unreachable by DEBUG, as an example, hard disc bootstrap sector as programmed in *New Zealand*, an alternative method is to compose a few lines in low level program constructs, utilizing DEBUG, to deliver proper *BIOS* (basic input output system) low-level commands to examine sectors being referred to. The sectors may be output to a pipe, utilising DEBUG, or dissected promptly. The instructions sequence below, written using DEBUG, beginning at memory address 100h scans the fixed disc *boot sector* into working storage utilizing BIOS low-level command INT 13h function 02h. The ES:BX needs to reference the working storage address to store the contents of the disc *sector* for the BIOS function to correctly operate. In the program listing of Fig. 7 on Intel machine or its classes, ES contains identical quantity with DS while BX point at address 800h in the referenced data memory

```
MOVAX, DS
MOV ES, AX
MOV AX,0201 ; service 02h, 1 sector
MOV CX,0001 ; track 0, sector 1
MOV DX,0080 ; head 0, drive 0 (i.e., first hard disk)
MOV BX,0800
INT 13h                 ; BIOS routine
JMP 10E                 ; halt here
```

Figure 7: Sample program listing of a virus disassembly process

Running the program in Fig. 7 by typing G 10E, inserting the program watch at address 10E, contents of address DS:0800 may either be listed on the console or directly analysed. Encrypted viruses present a slightly greater challenge, because they must be decrypted before they're disassembled. At times, this can be a bit crafty, since the virus author might have made use of anti-DEBUG measures as found in *Cascade*. Having converted the virus machine code to human readable, mnemonic form and listed onto a sink, examination of the mnemonics would show the way viral program performs, the effect and the way it spreads. An individual ought to naturally have accessible desktop computer documentation, which incorporates lists of interrupts and then painstakingly work over the disassembly, noting statements, operating system communication signals as well as working storage addresses. The representation should then begin to unfold. The self-reproducing portions of the viral code are thus confined with its payload. Any payload trigger conditions must be examined thoroughly, because they are easily misinterpreted.

After this process, a hex pattern can be extracted, which may be used to look for the virus; 16 bytes are usually enough, on the condition that the hex pattern is cautiously picked in order that it is a representation of a reasonably distinctive statements sequence, not likely present in any runnable program files.

4,3 Virus mutations and pattern-checking programs

Pattern-checking programs depend on looking for a pattern recognised to live within a virus. A version of a virus may be maliciously programmed such that it would not be recognised by a pattern-checker. The sequence of instructions that are independent, could be changed or an equivalent effect could be implemented by making use of various statements. For instance:

$MOV\ AX, 7f00h$
$MOV\ BX, 0$

Inside an infectious code might be rearranged as:

$MOV\ BX, 0$
$MOV\ AX, 7f00h$

A string-matching program searching for the string generated through the initial statements list, $B800\ 7FBB\ 0000$, wouldn't accept the changed order $BB00\ 00B8\ 007F$. At times, the mutations of a living virus will be very comprehensive that the virus will bear only a small similarity to the first. $Hex\ strings$ drawn out of the first may not likely be within the later viral code. The virus, $Fu\ Manchu$ [13], presents similar a vast alteration of $Jerusalem$, to the extent of being classified as a different viral program.

4.4 Identification of a Dissected Virus

In order to establish a positive identification of a *parasitic virus*, set up a dirty PC and make two copies of the same *experimental* executable. Infect one executable using the newly discovered virus and the other using the original virus. Run the DOS COMP command. If there are no differences, the virus is the same as the one already captured. If differences are discovered, the virus could still be the same, but it could be encrypted or self-modifying. This will have to be analysed more closely using the method outlined in section 4.2.1. *Boot sector viruses* are similarly identified.

4.5 Managing Virus

Virus countermeasures can be broadly classified into two groups: *proactive* and *reactive* measures. *Proactive* measures include the use of **scanning** and **monitoring** software—in the nature of $terminate - and - stay - resident\ (TSR)$ programs. A major drawback of these TSRs is that a well-written virus program may effortlessly circumvent or disable them. The mechanism employed by TSRs [14] to divert secondary storage inputs and outputs, i.e., to vary the vectors in the operating system's table of communication signals, precisely mirrors the one employed in nearly all viral programs. In addition, tracking process negatively impacts throughput and is quite incongruous with distributed applications, certain software packages, etc.

Scanning software work on the principle of signature verification against known *database* collections. When a novel virus emerges, it's separated into its constituent parts for the purpose of examining each part while a distinctive hex string of between 10 and 16 characters is saved to a databank. A tokenizer software then scrutinises all runnable program files on the secondary storage as well as OS, including the bootstrap sector, matching their contents with identified malware patterns. However, the demerits of such an approach is readily apparent: this sort of program is merely effective against identified malware patterns. consequently, the database must frequently be refreshed with novel patterns upon the release of new malware.

Another problem with the *scanning* software is the length of the virus pattern. If a short pattern is used, chances are that the scanning software will produce a number of false positives, finding the pattern in completely innocuous software. If a long pattern is used, false positives will be reduced, while incidence of false negatives will be on the increase since any mutation of a virus will have a better chance of not matching the pattern. This is especially true of encrypted viruses. Most scanning software is slow, although it is possible to speed the search for viruses by not looking at all locations, but only at the locations known to be infected by a specific virus. Unfortunately, this complicates both the scanning software and the list of viruses, which become more difficult to keep up to date and absolutely correct, thus increasing the risk of false negatives. It is because of this that this approach may be found to be appropriate only in special circumstances.\

Reactive measures largely make use of checksumming algorithms. The idea is that calculation of a checksum is performed on all executables on the computer then periodic recalculations are performed so as to be sure the fingerprint is intact. Should a malware infect a runnable program, it will alter not less than one bit inside the program file; a situation that can give a totally unrelated fingerprint. Even though this seems rather reactive, since a virus attack will be discovered after it happened, it could be made proactive with slight adjustment in the implementation.

The condition here is that prior to performing the initial checksum calculations, all executables are assumed to be 'clean' that is, virus-free. Installing all software from the manufacturers' original disks after performing a *hard* format of the fixed disc can do this. Then the checksumming algorithm is run on each of the executables to get a *fingerprint* or a *digital signature* of the executable. These *fingerprints* or *digital signatures* are stored in a database or individually stored in the executable file's header. Prior to running an executable on the system, a similar checksum is performed on the executable file and then compared with the initial checksum. A discrepancy in either of the checksums means that a virus has hit the executable. The system may be programmed to take any action for instance, ringing the system bell for a specified length of time, displaying a prompt in a particular font and colour or both, in order to alert the user. The system may then be shut down, as a way of getting rid of the malware from the working storage, install a clean copy of the infected executable and another checksum of the executable calculated and stored. This saves the system from being infected with a virus.

It is, however, assumed that the fingerprint algorithm is computationally intractable such that its results are not compromised by a malware thus making it vulnerable to infection both in the short and long term. This requirement makes utilisation of a *cryptographically strong* checksumming algorithm imperative, eliminating the other two approaches namely: simple checksums and cyclic redundancy checks (CRCs). This assumption calls for the use of the so-called *one-way trapdoor function* [15] because of its mathematical properties. The checksumming approach, as outlined here, happens to the singular fail-safe approach that discover every malware, now and in the time ahead, deterministically. A relatively permanent plan for malware defence is the use of a *cryptographically strong checksum* algorithm.

5. CONCLUSION

Proper documentation of all the known executable paths on a computer system formed an integral part of the methodology of this paper. The design principle focused on both virus-specific and non-specific algorithms. While the concept of virus-specific algorithms is appealing and somewhat straightforward in design, keeping the database of known virus patterns up-to-date made the approach non-beneficial in the long term. Software solutions could be obsolete almost as soon as they were procured because of the emergence of new viruses, or mutations of existing ones, on the scene. For a long-term foolproof protection against present and future viruses, a virus non-specific algorithm is favoured. Though the design and implementation were more rigorous and involved, the *cost-benefit* analysis more than justify these initial hurdles. A determining factor in a real-life scenario, however, is the prevailing exigency.

REFERENCES

[1] Lance J. Hoffman, editor (1990). *Rogue Programs: Viruses ,Worms, and Trojan Horses*. VanNostrand Reinhold, New York, NY.
[2] Leonard Adleman (1990). An abstract theory of computer viruses. In *Lecture Notes in Computer Science, vol 403*. Springer-Verlag.
[3] Alan Solomon (1991). *PC VIRUSES Detection, Analysis and Cure*. Springer-Verlag, London.
[4] David Ferbrache (1992). *A Pathology of Computer Viruses*. Springer-Verlag.
[5] Eugene H. Spafford. An analysis of the internet worm. In C. Ghezzi and J. A. McDermid, editors, *Proceedings of the 2nd European Software Engineering Conference*, pages 446–468. Springer-Verlag, September 1989.
[6] Peter J. Denning, editor (1990). *Computers Under Attack: Intruders, Worms and Viruses*. ACM Press (Addison-Wesley).
[7] Donn Seeley (1990). Password cracking: A game of wits. *Communications of the ACM*, 32(6):700–703.
[8] Yisrael Radai (1991). Checksumming techniques for anti-viral purposes. *1st Virus Bulletin Conference*, pages 39–68.
[9] Brown, Ralf and Jim, Kyle (1991). *PC Interrupts: A Programmer's Reference to BIOS, DOS and Third Party Calls,* Addison-Wesley.
[10] Jan Hruska (1990). *Computer Viruses and Anti-Virus Warfare*. Ellis Horwood, Chichester, England.
[11] Eugene H. Spafford, et al (1989). *Computer Viruses: Dealing with Electronic Vandalism and Programmed Threats*. ADAPSO, Arlington, VA.
[12] Eugene H. Spafford (1991). Computer viruses: A form of artificial life? In D. Farmer, C. Langton, S. Rasmussen, and C. Taylor, editors, *Artificial Life II, Studies in the Sciences of Complexity*, pages 727–747. Addison-Wesley, Redwood City, CA. Proceedings of the second conference on artificial life.
[13] Sandeep Kumar and Eugene H. Spafford (1992). A generic virus scanner in C++. In *Proceedings of the 8th Computer Security Applications Conference*, pages 210–219, Los Alamitos CA. ACM and IEEE, IEEE Press.
[14] Duncan Ray, Editor (1988). The MS-DOS Encyclopaedia. *Microsoft Press.*
[15] Odule Tola John (2007). Incremental Cryptography and Security of Public Hash Functions. *Journal of the Nigerian Association of Mathematical Physics,* 11, 467-474.