available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Securing communication using function extraction technology for malicious code behavior analysis

K. Vimal Kumar

SRM Institute of Management and Technology, Ghaziabad, India

ARTICLE INFO

Article history:
Received 20 June 2008
Accepted 30 September 2008

Keywords:
Malicious code
Function extraction technology
Virus
Computer threats

ABSTRACT

Since computer hardware and Internet is growing so fast today, security threats of malicious executable code are getting more serious. Basically, malicious executable codes are categorized into three kinds – *virus, Trojan Horse, and worm*. Current anti-virus products cannot detect all the malicious codes, especially for those unseen, polymorphism malicious executable codes. The newly developed virus will create the damages before it has been found and updated in database. The basic idea of the proposed system is, it will analyze the behavior of the malicious codes and based on the behavior signature of the malicious code content filtering mechanism will be used to filter out contents, so that, the system will be secured from the future communication processes. The behavior of the code is analyzed using the function extraction technology. The function extraction technology will replace the function codes into algebraic expressions. Based on the behavior of the malicious codes, it will be categorized into different kinds of malicious codes. The detected malicious code will be prevented from execution. Based on the type of malicious code, appropriate security mechanism will be used for further communication.

1. Introduction

In this present world of fast communication, malicious attacks on systems are a threat to business, government, and defense. One of the results of increasing number of terrorist attacks throughout the world as well as the seemingly frequent release of new malicious code has been the realization that better mechanisms need to be developed to share information (White and Dicenso, 2005). Many of the malicious attacks exploit system behavior unknown to the developers who created it. In today's state of art, software engineers have no practical means to determine how a sizable program will behave in all circumstances of use. This leads to many problems in security and survivability. If full behavior of a code is unknown, there will be too many embedded errors, vulnerabilities, and malicious code. The task of understanding program behavior is a haphazard, error-prone, resource-intensive process carried out by programmers and analysts in human time scale. Yet reliable understanding is essential to discover vulnerabilities and malicious code. Moreover, attackers can make deleterious modifications to programs at any time; this makes the task of behavior discovery a never ending process. The problem clearly exceeds manual capabilities and must be addressed through automation. So, the need for automatic behavior analysis technique increases (Mcgraw and Morrisett, 2000). The function extraction technology treats programs as rules of mathematical functions, that is, mappings from domains (Input, stimuli) to ranges (outputs, responses). It does not depend on subject matter that the program deals with.

Basically, malicious executable codes are categorized into three kinds. The first is called virus, which always infect other benign programs. The second is called Trojan Horse which always masquerade its malicious executable code inside

E-mail address: vimalkumar_k@fastmail.fm
0167-4048/$ – see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2008.09.006

a useful utility or freeware program (Deng et al., 2003). The third type of malicious code is called worm, which can replicate and distribute itself automatically around the network. The signature based malicious codes will check for a code string in the code and if it exists, it will be designated as malicious. Since there are attackers who modify the code strings slightly to make the detection process bit difficult. So, the need for behavior based detection increases drastically (Geer, 2006). Using the behavior analysis process, the malicious codes can be categorized into its types (as mentioned above). The newly detected malicious codes will be updated in the virus database for offline use and the appropriate security mechanism will be enabled for securing the future communication process. The chosen security measure for this system is content filtering. The content filtering process needs to match the behavior signature that is present in virus database, with that of the signature generated for the code that has been received. If there is a match, then the system will discard the code.

2. Related work

In today's state of art, there is no practical means to determine how a sizable program will behave in all circumstances of use. This leads to the heart of many problems in security and survivability. If full behavior is unknown, too many are embedded errors, vulnerabilities, and malicious code. To understand the program behavior, the function-theoretic foundations approach for automatic calculation of behavior is used. These foundations treat program control structures as mathematical functions or relations (Pleszkoch and Linger, 2004).

There are systems, which will perform the static analysis and dynamic analysis processes for detecting the malicious code. This test bed is called as Malicious Code Test bed (MCT) (Lo et al., 1991). The existing systems can detect the malicious code by analyzing the binary code of the malicious code (Bergeron et al., 1999). In such systems, the binary code has been converted into high-level language code and then analyzed in parts. In some other systems, the malicious code can be identified using the signature. The virus signature will be stored in the virus database and all the codes will be analyzed for a pattern that has been specified in the database. If there is such a pattern, then the system will designate that piece of code as malicious. The virus database has to be updated frequently for new malicious code detection. The drawback of this system is the malicious attack will occur before the database has been updated for new malicious code. There are systems that can detect and prevent the execution of malicious code. Those kinds of systems use an architectural technique called Run-time Execution Monitoring (REM), which will monitor the program execution to detect flow anomalies (Murat Fiskiran and Lee, 2004). But, the disadvantage is it can detect flow anomalies that occur only during execution process. The existing system can detect the malicious codes by executing it in controlled environment. There are certain systems that can continue to function in case of an attack on a component that it is trying to use. This kind of failsafe component will perform the integrity validations and checks on a requested component. This system also focuses on the integrity check on the original component and its add-ons to detect possible malicious codes added in an unauthorized manner. But, it does not detect the functionality of the code (Park et al., 2006).

3. Proposed system

The proposed system will overcome the limitations of the existing system and it also includes some additional features. The features of this system are the identification of malicious codes using function extraction technology. Based on the behavior signatures, the malicious code will be classified into its constituents. Once the malicious code has been identified, it will be prevented from execution. In order to provide protection to system when there is a newer attack of same form, the system uses the content filtering mechanism, which uses behavior signature generated during identification process. The content filtering mechanism uses the pattern matching process with the new packets for the behavior signature stored in the behavior catalog.

4. Product perspective

As shown in Fig. 1, if a system connected to a LAN is connected to internet, then it is more prone to malicious activities. Detecting the malicious codes and then preventing it can prevent these activities. This product has prevention of malicious codes as main aspect. This product will scan for any vulnerability in the entry of files into a system. This system will also replace the anti-virus software that being used. Normally, the anti-virus software will have the detection feature alone, but this product has the prevention feature also. This is an enhancement of the anti-virus products.

5. User classes and characteristics

The various user classes are – internet users, IT organizations. This user class does not need any additional education or experience to use this product. The frequent user of this product is the internet user. Since the malicious activities are stimulated by the use of internet, there is more need for this product to have their system more secured. The organization, which has more sensitive data, needs this product for securing information from malicious activity.

6. System features

The proposed system has the following features:

- Detection of newly developed malicious code
- Classification of malicious codes
- Prevention of malicious code execution
- Recovery from malicious code
- Enabling the required security measures

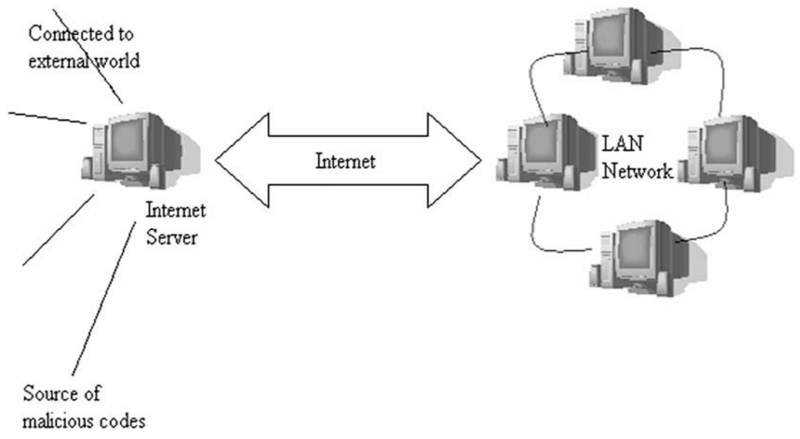


Fig. 1 – Overall diagram.

7. System architecture

The overall system architecture will be as shown in Fig. 2, which has the modules – behavior analysis and content filtering. The behavior of the program code will be analyzed and the signature will be updated in the database, for offline use also. The Behavior analysis uses the function extraction technology, which will analyze the functionality of the functions in the whole program (Pleszkoch and Linger, 2004). The different kinds of malicious codes have certain functionalities as major characteristics. So, the behavior analyzer will analyze the program and based on the functionality each function will be given a weight. The sum of these weights will be used for predicting the kind of malicious code. Once the database is updated, the content filters can use this virus database for filtering out the incoming packets. The content filters will generally look for a pattern in the packet's payload part (Whitman, 2003). This system uses the pattern matching process in the payload, which will be done only after verifying the packet's header part. Since the database will have category like address, pattern and also the level of attack, the content filtering can be applied only to those packets, which are not in high level of attack. This can be done by verifying the header part with the address stored in the database. This will reduce unnecessary pattern matching process. But, the pattern matching process is needed when there is a new kind of packet (which is not stored in the database).

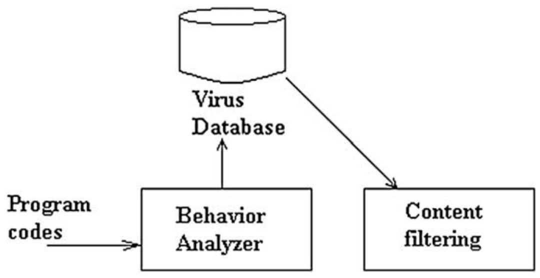


Fig. 2 – Overall block diagram.

8. Architecture for behavior analysis

Extractor is basically a function extractor, which will analyze the behavior or functionality of a code. The extractor is as shown in Fig. 3. The function extractor uses the simplification rules and abstraction rules, which are applied to the program code before function extraction process (Pleszkoch and Linger, 2004). Once the behavior signature generated for a malicious code, it will be updated in the behavior catalog. The behavior catalog will be used for the classification purpose. The malicious code will be classified based on the heuristic methods such as, weight-based method and rank based method.

In weight-based method, each functionality will be assigned a weight and after analyzing the whole program the summation of the weights will be used for designating the code as malicious or not. Similarly, the rank based method assigns ranks for each function identified.

9. Introduction to function extraction

The task of understanding program behavior is a haphazard, error-prone, resource-intensive process carried out by programmers and analysts in human time scale. Yet reliable understanding is essential to discover vulnerabilities and malicious code. And because attackers can make deleterious modifications to programs at any time, the task of behavior discovery never ends. The problem clearly exceeds manual capabilities and must be addressed through automation. The

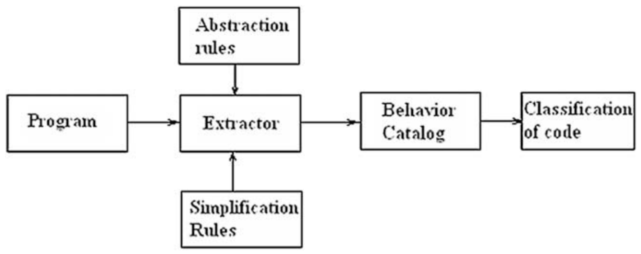


Fig. 3 – Block diagram for behavior analysis.

function extraction technology converts the program code into mathematical expressions. The stepwise refinements of the codes will finally give us the overall behavior of the code (Pleszkoch and Linger, 2004).

Consider the sequence control structure below constructed of assignments that operate on small integers *x* and *y*. The behavior extraction question asks: What does this program do, that is, what function does it compute? The answer is not obvious at first glance.

```
do
    $x := x - y$
    $y := y + x$
    $x := y - x$
end do
```

Function extraction requires deriving a procedure-free expression of what this structure does from beginning to end for all values of *x* and *y*. For a sequence structure, this requires composing the statements to determine their net, sequence-free effect. A simple trace table as depicted below, in Table 1, can be used for this purpose, with a row for every assignment and a column for every data variable assigned. Cells in the table record the effect of the row assignments on the variables. Subscripts are attached to variables to index effects from row to row, with 0 denoting initial values.

By substitution,

$$\begin{aligned} x3 &= y2 - x2 \\ &= y1 + x1 - x1 \\ &= y1 \\ &= y0 \end{aligned} \qquad \begin{aligned} y3 &= y2 \\ &= y1 + x1 \\ &= y0 + x0 - y0 \\ &= x0 \end{aligned}$$

Thus, the sequence just swaps the data variables. This is the behavior of the code, which is then added as a comment to the code as shown below.

```
[x, y := y, x]
do
    $x := x + y$
    $y := x - y$
    $x := x - y$
end do
```

To analyze a very large sequence of code, the same procedure will be used for different parts of codes in a stepwise manner.

10. Program behavior signature

Traditional engineering disciplines depend on rigorous methods to evaluate the expressions (equations, for example) that represent and manipulate their subject matter. Yet the discipline of software engineering has no practical means to fully evaluate the expressions it produces. In this case, the expressions are computer programs, and evaluation means understanding their full behavior, right or wrong, intended or malicious. Short of substantial time and effort, no software engineer can say for sure what a sizable program does in all circumstances of use. Yet modern society is dependent on the correct functioning of countless large-scale systems composed of programs whose full behavior and security properties are not reliably known. Many of these systems control key infrastructures in communication, energy, finance, and transportation. The existence and malicious exploitation of unknown functionality is the Achilles heel of software. It is little wonder that systems experience an endless flood of errors, vulnerabilities, and malicious code with frequently serious consequences. The task of understanding program behavior is a haphazard, error-prone, resource-intensive process carried out by programmers and analysts in human time scale. Yet reliable understanding is essential to discover vulnerabilities and malicious code. Since the attackers can make deleterious modifications to programs at any time, the task of behavior discovery never ends. The problem clearly exceeds manual capabilities and must be addressed through automation. Sizable programs are hard to understand because they contain a huge number of execution paths, any of which may contain security exposures. Faced with massive sets of possible executions, programmers can often do no more that achieve a general understanding of mainline program behavior. There is simply no way to understand and remember it all in today's state of practice. The situation is illuminated by an argument of the open source software movement, that more people looking at code will find more security flaws. It is interesting to note there is no open source arithmetic movement, seeking more people to determine if sums are flawed. Society knows how to make sums correct and has automated the process. It turns out the same may be true of software. Function-theoretic mathematical foundations of software illuminate a feasible strategy to develop innovative automation to address security exposures. An opportunity exists to move from an uncertain understanding of program security properties laboriously derived in human time scale to a precise understanding automatically computed in CPU time scale (Pleszkoch and Linger, 2004).

The function-theoretic model of software treats programs as rules for mathematical functions, that is, mappings from domains (inputs, stimuli) to ranges (outputs, responses), no matter what subject matter programs deal with. The key to the function-theoretic approach is the recognition that, while programs may contain an enormous number of execution paths, they are at the same time composed of a finite number of control structures, each of which implements a mathematical function or relation in the transformation of its inputs into outputs. In particular, the sequential logic of programs can be composed of single-entry, single-exit sequence (composition), alternation (if–then–else), and iteration (while–do) control structures, plus variants and extensions. This finite property of program logic viewed through the lens of function theory opens the possibility of automated calculation

Table 1 – Simple trace table.

Operation	Effect on X	Effect on Y
$x := x - y$	$x1 = x0 - y0$	$y1 = y0$
$y := y + x$	$x2 = x1$	$y2 = y1 + x1$
$x := y - x$	$x3 = y2 - x2$	$y3 = y2$

of program behavior. Every control structure in a program has a behavior signature, which can be extracted and composed with others in a stepwise process based on an algebra of functions that traverses the control structure hierarchy. The behavior signature of a program represents the specifications or business rules that it implements. These concepts are the basis for function extraction (FX) technology.

11. Function-theoretic approach

The canonical forms of behavior signatures of the basic control structures can be expressed through function composition and case analysis as follows (for control structure labeled P, operations on data labeled g and h, predicate labeled p, and program function labeled f). These function equations are independent of language syntax and program subject matter, and define the mathematical basis for behavior calculation:

11.1. Sequence control structure

The program function of a sequence P: g; h can be given by

$f = [P] = [g;\ h] = [h] \circ [g]$

where the square brackets denote the behavior signature of the enclosed program and "o" denotes the composition operator. That is, the program function of a sequence can be calculated by ordinary function composition of its constituent parts as illustrated in the example above.

11.2. Alternation control structure

The behavior signature of an alternation control structure P: if p then g else h endif can be given by

$f = [P] = [\text{if } p \text{ then } g \text{ else } h \text{ endif}]$
$= ([p] = \text{true} \rightarrow [g] | [p] = \text{false} \rightarrow [h])$

where | is the "or" symbol. That is, the program function of an alternation is given by a case analysis of the true and false branches, and the opportunity to combine them into a single abstraction as in the maximum operation.

11.3. Iteration control structure

For iteration control structures, the program function is given by a mathematical analysis of the potentially infinite number of loop iterations using quantification over the natural numbers:

P: while p do g enddo
can be re-expressed as
$f = [P] = \{(s,t)\text{: exists finite } n \geq 0 \text{ such that } (s,t) \in ([p] = \text{false}) \circ ([g] \circ ([p] = \text{true}))^n\}$

Notice that this results in the function [P] being undefined on all initial states that cause the loop to fail to terminate. Fortunately, there is an alternative analysis that is very useful in practice, using function composition and case analysis in a recursive equation based on the equivalence of an iteration control structure and an iteration-free control structure (an if–then structure):

$f = [P] = [\text{while } p \text{ do } g \text{ enddo}]$
$= [\text{if } p \text{ then } g;\ \text{while } p \text{ do } g \text{ enddo endif}]$
$= [\text{if } p \text{ then } g;\ f \text{ endif}]$

Function f must therefore satisfy

$f = ([p] = \text{true} \rightarrow [f] \circ [g] | [p] = \text{false} \rightarrow I)$

where I is the identity function. Note that the converse is not true; there may be other functions that satisfy this recursive equation that are only subsets of f. However, even these subsets can prove useful in behavior calculation.

In addition to these function equations, key theorems of function-theoretic mathematics provide important guidance for the behavior calculation process (Linger et al., 2002).

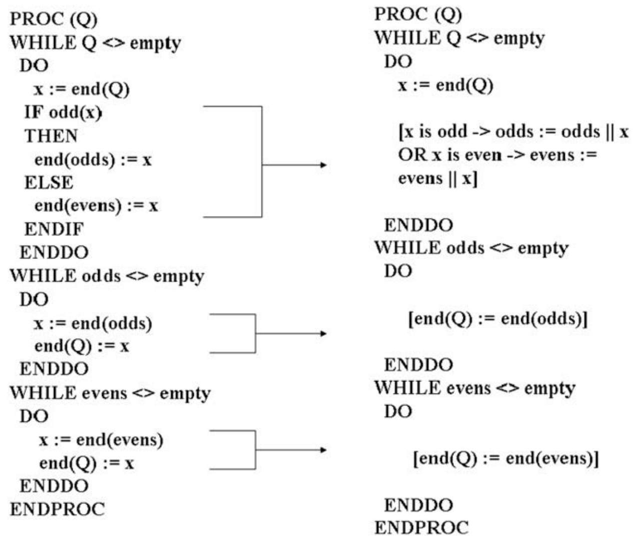
Logic Structure Theorem. *This theorem guarantees the sufficiency of sequence, alternation, and iteration control structures to represent any sequential logic. (Extensions and variants of these structures are included as well.) Thus, program logic can be expressed in nested and sequenced single-entry, single-exit structures, each with a common underlying mathematical model, namely, the function mappings defined above.*

Abstraction/Refinement Theorem. *This theorem and an associated Axiom of Replacement define conditions for substitution of behavior signatures and their control structure refinements, thereby enabling behavior extraction in a stepwise, algebraic process.*

Flow Verification Theorem. *This theorem defines conditions for correctness of control structures with respect to their behavior signatures. As noted above, even though programs can contain an enormous number of paths, they are expressed in a finite number of control structures, each of which can be verified in from one to three reasoning steps as defined by the theorem. Verification is thus reduced to a finite process.*

12. Stepwise behavior extraction

A behavior signature defines behavior identical to that of the control structure from which it was extracted, that is, the signature and control structure are function equivalent mappings of inputs into outputs. Thus, signatures can be freely substituted for corresponding control structures (Pleszkoch and Linger, 2004). Such substitution defines an algebra of functions that permits stepwise extraction of program behavior by traversing control structure hierarchies from bottom to top. At each step, net effects of control structures are composed and propagated while details are left behind. In illustration, consider the miniature program on the left of Fig. 4 and the question of what it does. The program takes as input and produces as output a queue of integers named Q, and uses local queues of integers named odds and



```
PROC (Q)
WHILE Q <> empty
 DO
   x := end(Q)
   IF odd(x)
   THEN
     end(odds) := x
   ELSE
     end(evens) := x
   ENDIF
 ENDDO
WHILE odds <> empty
 DO
   x := end(odds)
   end(Q) := x
 ENDDO
WHILE evens <> empty
 DO
   x := end(evens)
   end(Q) := x
 ENDDO
ENDPROC
```

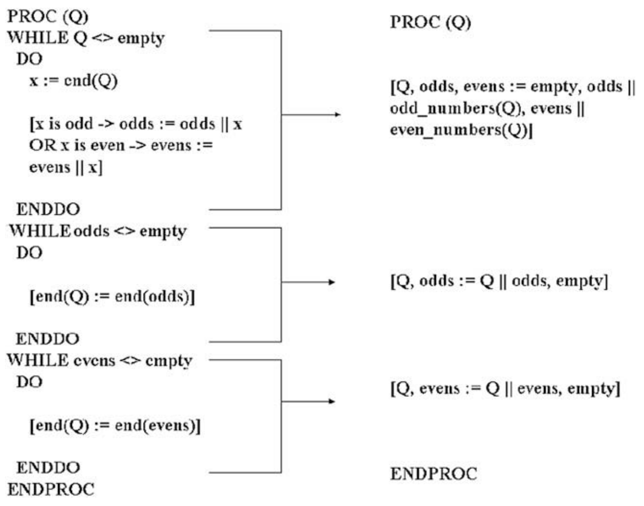
```
PROC (Q)
WHILE Q <> empty
 DO
   x := end(Q)

   [x is odd -> odds := odds || x
   OR x is even -> evens :=
   evens || x]

 ENDDO
WHILE odds <> empty
 DO

   [end(Q) := end(odds)]

 ENDDO
WHILE evens <> empty
 DO

   [end(Q) := end(evens)]

 ENDDO
ENDPROC
```

Fig. 4 – Stepwise extraction – first step.

evens and a local integer variable named x (declarations not shown). The symbol <> stands for not equal, || for concatenation. The stepwise behavior calculation process is depicted in Figs. 4–6.

In the first step, the interior conditional statement will be extracted out into expressions. Since, it is an IF–THEN–ELSE statement; the condition has to be extracted for signature generation. Both the statements for condition to be true and also for false will be generated in the signature. The next process will be to reduce the expressions into single expressions.

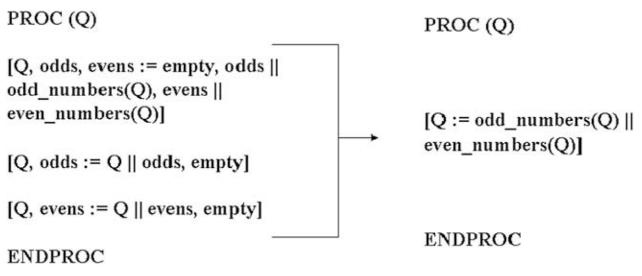
In the second step of the extraction process, the outer loop of the conditional statements will be taken into consideration. Since, the outer loop has a WHILE condition, here also the condition has to be extracted out. It is similar to IF–THEN–ELSE extraction process.



```
PROC (Q)
WHILE Q <> empty
 DO
   x := end(Q)

   [x is odd -> odds := odds || x
   OR x is even -> evens :=
   evens || x]

 ENDDO
WHILE odds <> empty
 DO

   [end(Q) := end(odds)]

 ENDDO
WHILE evens <> empty
 DO

   [end(Q) := end(evens)]

 ENDDO
ENDPROC
```

```
PROC (Q)

[Q, odds, evens := empty, odds ||
odd_numbers(Q), evens ||
even_numbers(Q)]

[Q, odds := Q || odds, empty]

[Q, evens := Q || evens, empty]

ENDPROC
```

Fig. 5 – Stepwise extraction – second step.



```
PROC (Q)

[Q, odds, evens := empty, odds ||
odd_numbers(Q), evens ||
even_numbers(Q)]

[Q, odds := Q || odds, empty]

[Q, evens := Q || evens, empty]

ENDPROC
```

```
PROC (Q)

[Q := odd_numbers(Q) ||
even_numbers(Q)]

ENDPROC
```

Fig. 6 – Stepwise extraction – final step.

In the final step of extraction process, the overall functionality of the program has to be extracted out. In this process, the signatures generated inside the program will be converted into single expression. The overall signature is as-built behavior specification, that is, the calculated behavior of the entire program. The extraction process reveals that the program creates a new version of queue Q, now containing its original odd numbers followed by its original even numbers. In this process the intermediate control structures and data uses drop out to simplify scale-up by subsuming their functional effects into higher-level abstractions. The principal behavior calculation process is function composition through value substitution, which by definition eliminates intermediate expressions at successive levels of abstraction. As noted above, programs can exhibit an enormous number of execution paths, but are comprised of a finite number of control structures, so the behavior calculation process is itself finite and guaranteed to terminate.

13. Database updation

The behavior signature of the malicious code has been obtained using function extraction technology. These behavior signatures have to be updated in the database for future verification process. Since the implementation language is other than java, the process of updation can be done using ODBC concept. Once the database has been updated, it can be used by the content filtering module.

The database should have the fields such as, behavior signature, URL address and also the action to be performed for that content. The database will be updated only when the code is found to be malicious.

14. Content filtering

A content filter is a software filter that allows administrators to restrict accessible content from within the network. Content filtering will analyze the payload of a packet for the filtering mechanism. The most common application of this is the restriction of web sites with non-business related material. The advantage of content filtering is the assurance that malicious content available materials can be restricted and these contents cannot waste organizational time and resources.

Content-based filtering, 'characterizes the contents of the message and the information needs of potential message recipients and then using these representations to intelligently match messages to receivers'. Research based systems, such as SIFT, and the commercial systems (e.g., DansGuardian, iProtectYou, Parental Filter, Symatec Web Security, and We-Blocker) usually use URL or keywords to characterize content, while the social filtering systems use personal organizational interrelationships of individuals in a community. Some researchers interpret this as a collaborative filtering approach, which is now being referred to as 'recommendation systems'. Filtering knowledge can be obtained implicitly or explicitly. The implicit approach utilizes the user's reaction to incoming data in order to learn from it. For example, time for reading messages, hyperlink clicking, document printing, and scrolling, are regarded as the user's interest expression. The explicit approach requires users to fill out a form describing their areas of interest or relevant parameters. There are many rule based filtering systems that support users to construct more flexible filters, for example Lens and ISCREEN (Churcharoenkrung et al., 2005).

Here, this system uses the implicit filtering process. When there is an operation to import a file into the system, the process of behavior signature extraction will be enabled. The generated behavior signature will be verified with the virus database for its existence. If it exists the file will be designated as malicious and will be blocked from access. If it is a new kind, then it will be updated in virus database. So, the future communication process will be secured.

In order to avoid unnecessary frequent signature extraction, the system initially checks the URL address of the content. The URL of the content will be checked with the database for its existence. If it exists, the system will take an appropriate action that is specified in the database.

15. Restoring the files

Malicious codes are a major problem in modern day computing. Generally, a malicious code is a program (or some unit of code, e.g., instructions to which the computer responds, such as a code block, code element or code segment) that may attach to other programs and/or objects, may replicate itself, and/or may perform unsolicited or malicious actions on a computer system. Although described herein as relating to computer viruses, the present disclosure may be applied to any type of malicious code capable of modifying one or more portions of a computer's resources. One cure for recovering from a computer virus may include removing the computer virus. This may include disabling the virus in an infected object, which may be, for example, a file, a memory area, or the boot sector of a storage medium. However, recent computer viruses have also been seen which manipulate objects in addition to the originally infected object, for example, by deleting or renaming files, manipulating system registry and initialization files, and/or creating unwanted services and processes.

Computer viruses have been seen that may rename an existing file on the computer system and/or replace it with a different file that causes the computer to operate in an undesirable manner. In addition, a virus may modify existing system configuration files while embedding itself in the computer system. An example of a computer virus that does both is the "Happy99.Worm" virus. This particular type of virus travels as an attachment to an email message and causes an infected computer to attach a copy of the virus to outgoing email messages. This type of virus may also place one or more hidden files on the computer's hard drive and/or make changes to the Windows registry file. For example, the "Happy99.Worm" virus renames the file "Wsock32.d11" to "Wsock32.ska" and replaces the original "Wsock32.d11" with its own version of the file. The "Happy99.Worm" virus also creates several other files on the computer system including "Ska.exe" and adds a line to the Windows registry file instructing the computer to run the "Ska.exe" file upon startup.

Simply disabling or removing the virus code without restoring or correctly renaming the files, etc., and/or removing unwanted services or processes, will not effectively restore the computer system. That is, restoring an object to which the virus has attached itself may not always be sufficient, particularly if a number of other objects have been created or modified by the computer virus.

Because each virus may affect different portions of a computer system, specific treatments are required and may require a number of operating system specific operations performed on any number of objects. Therefore, there is a need for a complete cure of an infected computer system that restores all the affected objects.

In order to restore files that may have been manipulated or damaged by a computer virus, a restoration command data file may also include "Delete File", "Rename File", and/or "Copy File" file system commands for manipulating files located on the computer system. In addition, a "Shell" command may be provided through which a system shell command may be executed. These commands may use one or more file names as input parameters depending on usage and return an error condition in case of failure. In case a file to be manipulated is currently used by the system and cannot be accessed, the file system command will not return an error condition; rather, the command will alert the computer system that a computer restart is necessary to release the file. Once the file is released, the present method and system will execute the previously attempted file system command.

Computer viruses may start unwanted processes and/or services running on a computer system. Accordingly, a restoration command data file may also contain process manipulation commands for stopping processes and services currently running on a computer system. For example, a "Kill Process" command may be used to stop a process currently running on the computer system, and a "Kill Service" command may be used to stop a service and remove it from the Windows registry file.

16. Conclusion

The objectives of behavior computation are to reveal the behavior signatures of vulnerabilities and malicious code as well as other security properties, and to provide a catalog of all program behavior for assessment of security properties and risks. The syntactic scanning of programs deal with surface features, and cannot get at the underlying semantics that reveal the function and intent of malicious code attacks. Because the functionality of a particular malicious code attack can be programmed and disbursed in any number of syntactic forms, syntactic detection can be difficult. However, all such forms will result in the same behavior signature at the semantic level, thereby reducing the deceptive power of syntactic variations. The behavior analysis process for malicious codes has generated the behavior signature of the program codes (both genuine and malicious code).

The module for identification of malicious code uses these behavior signatures as input and identifies the malicious code using the weight-based method. Once the behavior signature has been generated, it will be used by the content filtering process for preventing the malicious code's entry into the system. For achieving this, the database has to be updated by new signatures. Since, the files affected by malicious code can't be used further, it has to be restored back once again. This project overcomes the issues such as, identification of polymorphic malicious codes, updation of newly identified malicious code's signature in the virus database even when the system is in offline.

17. Future work

The limitation of this system is it cannot handle malicious codes in executable form. This can be overcome by splicing the binary code and converting it into high-level language. Once it has been converted, the code can be analyzed using this system. The security measure used here is content filtering alone. The system can be enhanced with the addition of some more security measures.

REFERENCES

Bergeron J, Debbabi M, Erhioui MM, Ktari B. Static analysis of binary code to isolate malicious behaviors. In: Proceedings of the IEEE eighth international workshop on enabling technologies. Infrastructure for Collaborative Enterprises; 1999 [WET ICE '99].

Churcharoenkrung N, Kim YS, Kang BH. Dynamic web content filtering based on user's knowledge. In: Proceedings of the international conference on information technology: coding and computing (ITCC'05); 2005.

Peter Shaohua Deng, Jau-Hwang Wang, Wen-Gong Shieh, Chih-Pin Yen, Cheng-Tan Tung. Intelligent automatic malicious code signatures extraction. In: Proceedings of the IEEE 37th annual 2003 international carnahan conference on security technology; 2003.

Geer David. Behavior based network security goes mainstream. IEEE Computer March 2006.

Linger R, Pleszkoch M, Walton G, Hevner A. Flow-service-quality engineering: foundations for network system analysis and development, CMU/SEI-2002-TN-01. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University; 2002.

Lo R, Kechen P, Crawford R, Ho W. Towards a test-bed for malicious code detection. IEEE; 1991.

Mcgraw Gray, Morrisett Greg. Attacking malicious code: a report to the Infosec Research Council. IEEE Software 2000.

A. Murat Fiskiran, Ruby B. Lee. Runtime Execution Monitoring (REM) to detect and prevent malicious code execution. In: Proceedings of the IEEE International Conference on Computer Design (ICCD'04); 2004.

Joon S. Park, Joseph Giordano, Gautam Jayaprakash. Component integrity check and recovery against malicious code. In: Proceedings of the 20th international conference on advanced information networking and applications (AINA'06); 2006.

Mark G. Pleszkoch, Richard C. Linger. Improving network system security with function extraction technology for automated program behavior. In: Proceedings of the 37th Hawaii international conference on system sciences; 2004.

Gregory B. White, David J. Dicenso. Information sharing needs for national security. In: Proceedings of the 38th Hawaii international conference on system sciences; 2005.

Whitman Michael E. Principles of information security. Vikas Publications; 2003.

K. Vimal Kumar, working as a Lecturer in Department of Computer Science & Engineering at SRM University, NCR Campus, Ghaziabad. I'm a Member of International Association for Computer Science & Information Technology (IACSIT). I have completed M.E. in Computer Science and Engineering from College of Engineering, Guindy (Anna University). My specializations are Network Security, Data Structure & Algorithms.