Obfuscated Malware Detection Using API Call Dependency

Chinmaya Kumar Patanaik
Dept. of CSE
IIT Guwahati, INDIA
chinmayapatanaik@yahoo.com

Ferdous A Barbhuiya
Dept. of CSE
IIT Guwahati, INDIA
ferdous@iitg.ernet.in

Sukumar Nandi
Dept. of CSE
IIT Guwahati, INDIA
sukumar@iitg.ernet.in

ABSTRACT

Malwares pose a grave threat to security of a network and host systems. Many events such as Distributed Denial-of-Service attacks, spam emails etc., often have malwares as their root cause. So a great deal of research is being invested in detection and removal of malwares. Thus many malware detection systems or antivirus softwares have come up. But the drawback of these antivirus softwares is they rely upon signature matching approach for malware detection which can be easily defeated using simple code obfuscation techniques. This has given rise to a new generation of metamorphic and polymorphic malwares. In this paper we proposed the approach of monitoring interdependent system calls to detect obfuscated malicious programs. We took some sample malwares and some common obfuscation techniques. We tested these obfuscated malwares against open source antivirus ClamAV and our detection model. The results obtained have been elaborated further in the paper. Again how our algorithm is sound against many drawbacks of the API call monitoring approach such as API call reordering, garbage API call insertion etc., are also described.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Security

Keywords

Malware, Obfuscation, API calls

1. INTRODUCTION

Malware is a category of malicious code that includes viruses, worms, and trojan horses etc. They normally utilize popular communication tools to spread, including worms sent through email and instant messages, virus-infected files downloaded from peer-to-peer connections etc. Malware also try to exploit existing software vulnerabilities and systems making their entry quiet and easy. They often carry out tasks such as click frauds, stealing passwords, changing the system configuration, using a system as a bot to launch further attacks etc. Now a days malwares like *Stuxnet* have been evolved which can even shut down a nuclear plant [1]. As per Falliere and Murchu in [15], Stuxnet used 4 zero day vulnerabilities in windows and 1 in scada systems to successfully launch its attack. Malwares often carry a payload which is triggered on host systems during execution or when a certain condition is met.

To overcome this security threat many network based and host based intrusion detection systems or antivirus softwares have been created. For detecting malwares many static and dynamic analysis techniques have been developed with each having their own shortcomings. In case of traditional antivirus softwares, they use signature matching approach to detect malwares [2]. They usually take a sequence of bytes, a string or file hashes in a malware which is unique to it and use that as its signature. So while scanning it actually looks for these specific malicious byte sequences or signatures in a program. Though the signature matching approach is quite fast and effective but it is not reliable. It has been proven completely helpless against zero-day attacks. Now a days with the evolution of metamorphic and polymorphic viruses, signature matching approach is proved to be useless as described in [17]. According to a report by Symantec, most of the viruses discovered each day are just obfuscated variations of the existing viruses. Virus writers use simple code obfuscation techniques to thwart signature matching. To counter the obfuscation techniques, different approaches have been proposed. For example Bruschi, Martignoni and Monga have proposed the control flow graph matching method in [8]. Here the CFG of a program is compared with the CFG of a particular section of a virus. If matching is found then it is proven to be malicious. Some other popular methods used to detect malwares are semantics based detection, call graph matching approach etc. With obfuscation techniques, though the structure of code sequence changes but their semantics remains the same. So using the semantic approach can be very helpful in detecting malicious programs. In case of call graph matching approach,system calls issued by a program are monitored and a call graph is generated. This call graph is then matched or its similarity is calculated with the call graph of a ma-

SecurIT'12, August 17-19, 2012, Kollam, Kerala, India

licious program. Though this approach is efficient for malware detection but it has some certain drawbacks. Using independent API call reordering or garbage API calll insertion, it can be easily defeated. In another approach for polymorphic virus detection, code emulators are deployed by antivirus softwares where virus is executed and monitored in an emulated environment. So at some point of time the decrypted virus body can be found in memory. Then signature matching is used to successfully detect them. So far many static analysis techniques have been proposed to detect obfuscated malwares. Apart from those many dynamic analysis approaches are also being used. But they had some significant shortcomings. Here in this paper we present how API call monitoring approach can be used efficiently to detect obfuscated versions of a malware.

The rest of the paper is organized in the following manner. In section 2 some existing methods are described followed by highlighting some common obfuscation techniques used for deceiving antivirus software in section 3. In section 4 we proposed our scheme to detect obfuscated malwares. In section 5 we showed some of our experimental results and finally concluded this paper in section 6.

2. PREVIOUS WORK

Our work is generally based on system call monitoring techniques to verify security properties of a program. At first we studied different kinds of obfuscation techniques currently used by metamorphic and polymorphic malwares. Christodorescu and Jha in [9] showed some common obfuscation techniques on certain malwares and tested those obfuscated malwares against some commercial antivirus scanners and their detection model. They used static analysis method to detect the obfuscated samples. They took the section of instructions of a virus which is used by different antiviruses as its signature. Then they made a code automaton of it. Now for any obfuscated version of that malware, its annotated control flow graph was made. Their main goal was to search for code section satisfying the malicious code automaton within the annotated control-flow graph of the program. Lee, Jeong et al. in [14] have dealt with metamorphic viruses by using code graph approach. Here the authors have proposed a scheme based on API calls issued by a program. They monitored the system calls issued by a malicious program and built a call graph. In the graph API calls were taken as nodes and according to jump instructions in the disassembled phase, edges were formed. They further unified those system calls based on their properties to certain groups and built a code graph. These code graphs were used as signatures for comparison. Now for each process they built separate code graphs and checked their similarity to detect malicious patterns.

But our work is closely related to work done in the following papers. In [11] Christodorescu, Jha et al., took the part common in different variants of a malware family and made a template comprising of instructions representing that character. For example in case of email worms they took the code section where it propagates via mails. Now they used def-use chain approach to find a block of instructions in a program satisfying the behavior showed by the template. They tried to find a block of instructions in an obfuscated sample that is semantically equivalent to the sequence of instructions in the signature of the original virus thereby detecting its maliciousness. Kolbitsch, Comparetti et al. in [12] used the behavioral approach to detect malwares. First they executed the sample malware in an emulated environment. Next they built a model that characterizes its behavior using interdependent system calls called behavior graph. The arguments of every the system call from the graph were noted. Now for each argument they back traced to find the source and instructions which effected the value of that argument and represented those instructions by a function. In case of any unknown program, their scanner tries to match the system calls invoked by the program with the behavior graph. If matching is found, then it treats that program as malicious. The authors have mentioned that this approach is unaffected by independent API call reordering, garbage API insertion techniques, obfuscation techniques.

In our paper we took the concept of observing certain interdependent system calls evoked by the malware and proposed a detection algorithm for obfuscated malware detection. The main difference between our scheme and the paper [12] is that we use dead end API calls(API calls with no further dependency) in the behavior graph to detect maliciousness. This is elaborated below. In case of an obfuscated version of that malware or any program we basically scanned for a sequence of those interdependent system calls . We tested these obfuscated malwares against our model and open source antivirus *ClamAV*. We also showed how our algorithm is robust to API call obfuscations.

3. OBFUSCATION OF MALWARES

Virus writers use many encryption techniques to escape detection. *Cascade* virus was the first DOS virus to use encryption technique. The virus starts with a constant decryption routine that is followed by the encrypted virus body. But those viruses were not that difficult to detect because they often had the same decryption loop in common. So those decryption loops were used as signatures of that virus. In case of oligomorphic viruses, with each new generation they add a new decryption routine. Polymorphic viruses can generate new decryption routine that uses new encryption methods. In case of polymorphism techniques, even though the virus is encrypted but its main body remains unchanged. So based on this idea dynamic decryption technique was used to successfully detect polymorphic viruses. But now a days polymorphic viruses are generated with Entry-point obfuscation characteristic. In case of Entry-point obfuscation, the entry point of the virus code is placed inside the program's code instead of being placed at the beginning or ending. Then unconditional or conditional jump statements are used to reach the entry point and execute the virus code.

Again in case of metamorphic viruses apart from encryption, they change their code whenever they replicate. They use some of the common code obfuscation techniques to change and recompile their code if a compiler is available at the system. So at each time their byte sequence pattern changes and it simply becomes impossible to detect with the current byte sequence matching approach. So the main difference between polymorphic and metamorphic viruses is that metamorphic viruses don't use decryption loops or a constant body. For our case we used the worm Bagle.A. We opened the virus in IDAPRO disassembler [3] and performed

behavior and code analysis. Rozinov has described the code analysis results in [16].

3.1 Virus Under Consideration: Bagle

Bagle was found in early 2004. Its main method of propagation was through emails. Upon execution for the first time it checks the system time for not later than January 28, 2004 or else it deletes itself. It makes a copy of itself to be used in mails. It checks whether it was launched from bbeagle.exe in system32 or else it copies itself to system32 and launches calc.exe to disguise itself. It then searches files with extensions *.wab*, *.txt*, *.htm*, and *.html* in local drives for email addresses and if found sends a copy of itself to those email addresses. To test the effectiveness of our detection algorithm we tested it against open source antivirus scanner *ClamAV*. In ClamAV the following sequence of bytes is used as signature for our sample Worm.Bagle.A.

```
3c25733e0d0a005243505420544f3a.........657865.
```

We opened the worm in *Ollydbg* and took out a particular section occurring within these sequence of bytes [4]. Then we performed many code obfuscation techniques on this particular section. The byte sequence of the section we took and its instruction sequence are given in tables 1 and 2 respectively.

Table 1: Signature

5241	4E44	255D	0064
6464	272C	2720	6464
204D	4D4D	2079	7979
7920	0048	483A	6D6D

Table 2: Instruction sequence of Worm.Bagle.A

52	push edx
41	inc ecx
4E	dec esi
44	inc esp
25 5D006464	and eax,6464005D
64:27	daa
2C 27	sub al,27
206464 20	and byte ptr ss:[esp+20],ah
4D	dec ebp
4D	dec ebp
4D	dec ebp
2079 79	and byte ptr ds:[ecx+79],bh
79 79	jns short Bagle-ge.00405847
2000	and byte ptr ds:[eax],al
48	dec eax
48	dec eax
316D 6D	cmp ch,byte ptr ss:[ebp+60]

3.2 Different Obfuscation Techniques

As said earlier metamorphic malwares frequently use code obfuscation techniques to avoid detection. We basically used the different obfuscation techniques described in [9, 13, 18]. Here we used the code obfuscation techniques in disassembly phase of a program.

3.2.1 Garbage Code Insertion

During evolving sometimes metamorphic viruses insert junk piece of code into their code sequence. Overall garbage code has no impact on the execution of the program or the function of the program. The main objective in using garbage code is upon insertion, it changes the byte sequence of the program. So traditional antiviruses which solely depend upon byte sequence matching approach often fail to detect the obfuscated malwares. This is one of the simplest obfuscation techniques. One simple example of garbage code insertion is to insert *NOP* instructions in between the code. They change execution state of the program only to change it back again. Use of *NOP* instructions and some other garbage instructions is displayed in table 3 on our sample bagle virus.

Table 3: Obfuscated instruction sequence of Worm.Bagle.A

52	push edx
41	inc ecx
4E	dec esi
44	inc esp
25 5D006464	and eax,6464005D
64:27	daa
90	nop
90	nop
2C 27	sub al,27
206464 20	and byte ptr ss:[esp+20],ah
90	nop
4D	dec ebp
4D	dec ebp
4D	dec ebp
44	inc esp
4C	dec esp
2079 79	and byte ptr ds:[ecx+79],bh
79 79	jns short Bagle-ge.00405847
2000	and byte ptr ds:[eax],al
48	dec eax
48	dec eax
316D 6D	cmp ch,byte ptr ss:[ebp+60]

Table 4: Newly generated signature after NOP insertion

5241	4E44	255D	0064
6464	2790	902C	2720
6464	2090	4D4D	4D44
4C20	7979	7979	2000
4848	3A6D	6D	

3.2.2 Register Usage exchange

Sometimes during mutation metamorphic viruses change the way registers and other memory variables are used in its code. One common technique is that in live range if we alter the use of registers then it won't have any overall effect but it will change the byte sequence signature of the virus. For example if the register *ecx* is dead in a live range than we can replace any other register or variable which is being used in that range with *ecx*. This method does not pose any challenge because except from certain register bytes, rest of the byte sequence still remain the same i.e. majority of the byte sequences remain same. So AV scanners using wild card will be able detect this kind of obfuscated malware.

3.2.3 Code Transposition

The main objective of code transposition technique is to alter the syntactic structure of the virus body. They alter the structure of codes in the virus but maintains the flow of execution. Insertion of unconditional branches is one of the easiest code transposition techniques. So in the end the obfuscated and the original sample both have the same execution flow but in case of the obfuscated sample, its instructions are just shuffled. In another case the functions of a program can be reordered using either conditional or unconditional jumps. It has the same effect as that of instruction reordering within a code section. Another method of code transposition technique is to change the position of independent instructions. It can be thought of randomly placing instructions which are not interdependent. In our case we only implemented the first kind of code transposition technique. The results are as shown in table 5 .

Table 5: Obfuscated instruction sequence of Worm.Bagle.A

	push edx
	inc ecx
	dec esi
	inc esp
	jmp S1
S2:	sub al,27
	and byte ptr ss:[esp+20],ah
	dec ebp
	jmp S3
S1:	and eax,6464005D
	daa
	jmp S2
S3:	dec ebp
	dec ebp
	and byte ptr ds:[ecx+79],bh
	jns short Bagle-ge.00405847
	and byte ptr ds:[eax],al
	dec eax
	dec eax
	cmp ch,byte ptr ss:[ebp+60]

3.2.4 Instruction Substitution

In comparison with the above obfuscation techniques, instruction substitution can be considered as very hard to detect. Here in position of a single or block of instructions, another block of instructions are placed which does the same thing. So overall though instructions of a particular section are changed but both the malware and its obfuscated version semantically perform the same operation. The reason they are often hard to detect is that it uses a library of instructions to replace or substitute a single instruction. We also used this technique on our sample bagle virus as shown in table 6.

3.2.5 Code Integration

This can be treated as one of the most sophisticated obfuscation techniques and very hard to detect. Win95/Zmist was the first virus which used this technique. In order to apply this technique, it inserts itself into an executable code and scatters its body among the instructions. Malicious fragments are then reconnected using proper flow transition instructions. The malicious code will be executed when the normal control flow reaches the first instruction of the virus code. As discussed earlier this can be treated as a case of Entry Point Obfuscation. The main threats posed by viruses using this technique is the problem of differentiating the actual malicious code which is being spread inside the benign code. That's why it can be considered as one of the most advanced obfuscation techniques [18].

Table 6: Obfuscated instruction sequence of Worm.Bagle.A

push edx
inc ecx
dec esi
add esp,01
and eax,6464005D
daa
sub al,27
and byte ptr ss:[esp+20],ah
sub ebp,03
and byte ptr ds:[ecx+79],bh
jns short Bagle-ge.00405847
and byte ptr ds:[eax],al
sub eax,02
cmp ch,byte ptr ss:[ebp+60]

4. PROPOSED SCHEME

In the above section some common obfuscation techniques used by metamorphic and polymorphic viruses have been shown with the help of our sample virus Bagle.A. So our problem statement comes down to detecting maliciousness of an obfuscated version of a known virus. In this paper we used an approach based on monitoring API calls made by the obfuscated sample. We took the help of *StraceNT* to obtain the system call trace of bagle.

To detect an obfuscated virus, the concept of going after semantics of a program as opposed to syntactic was proven to be very effective [11]. As we know API calls are the only way through which a program interacts with the operating system environment, so even though instruction sequence changes because of obfuscation, API call sequence will still remain the same. So we monitored the API calls issued by a program during execution. But due to obfuscation, arguments and the return values of API calls changed. To deal with this problem our algorithm will only look for interdependency between thecertain specific API calls. Because even though the arguments and the return values change, but the data dependency among them will remain intact. For example if we consider the case of Bagle virus, we can observe that the dependency array of the first API in the signature of Bagle contains the APIs GetModuleFileName, CreateFile, UnmapViewOfFile and GlobalFree from table 7. That means the return value of GlobalAlloc is used as an argument of these APIs. So in case of the obfuscated version, this dependency will also hold true. Therefore at first we took a particular virus and executed it in a controlled environment. Then we monitored all the system calls issued and picked out the system calls which were responsible for a specific behavior of that program. After that we checked the interdependency between those API calls i.e. whether the return value of any API call was used as an argument

by another API call. This trace of system calls with data dependency among them can be treated as the signature of that virus. For a better understanding we represented it in the form of a graph. In this graph we made each API call a node and according to data dependency among them we formed an edge. This can be called as a call graph or behavior graph because it shows a specific behavior of that virus. Now we inject this model of the virus into our detection routine along with any other benign programs or obfuscated version of that virus. In the detection routine our algorithm looks for the API calls as per the call graph, stores and further checks whether they follow the same dependency as per the dependency between API calls of the call graph. If found then it is detected as malicious. A detailed description of our proposed algorithm is given in the next section.

Christodorescu and Jha have described in their paper [10] that a malicious behavior should follow some particular specifications. Based on those rules we have the following assumptions.

4.1 Assumptions

1. The behavior must not constrain truly in-dependent operations. In our case this requirement is true. Because we have only taken the interdependent API calls describing the specific property of a malware. So independent API calls are not taken into consideration while building the call graph.

2. A specification must relate dependent operations. This also holds in our case as the API calls considered while making the call graph have data dependency among them.

3. A specification must capture only security related operations. In our case the API calls we took describes a specific property of a virus. So this condition also holds in our case. In case of bagle.a the API calls we took are used in making a base-64 encoded copy of the worm.

So in our sample case we took the original Bagle.A virus. Upon code analysis some of the obtained results are shown earlier. Bagle.a was disassembled using IDA pro. The function sub_4016CA is responsible for making a base 64 encoded copy of the virus. This copy is used as email attachments. Then we looked at the API calls being issued to perform this action. A graph showing the API calls along with their data dependencies represented by directed edges is shown in figure1.

Then we looked at their arguments and return values and made a trace of these system calls along with the interdependency among them. We treated this system call trace as the signature for Bagle.A virus. As a process can make at most one API call at a time so, a sequence number was given to each of the API calls according to their order of execution. The signature of Bagle.A is as shown in table 7.

Here it can be observed that there are some API calls which don't have any further dependencies. These API calls are

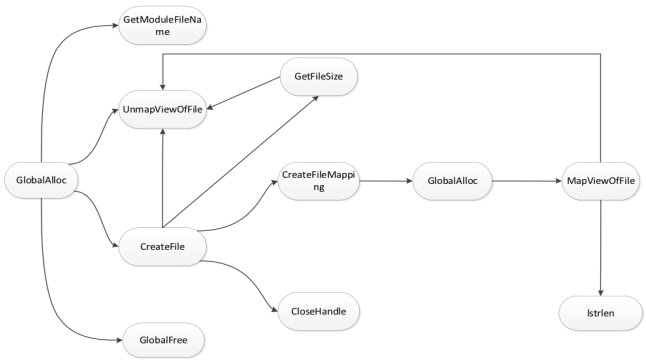

Figure 1: Call graph representation of Bagle.A

considered as dead ends. They have a specific role in our detection algorithm. This differentiates our scheme from the concept used in [12]. So the above API call sequence with shown dependencies is treated as the signature for Bagle.A virus. Now we took the traces of the obfuscated sample and injected them into our model along with the signature for Bagle.A to detect their maliciousness. Apart from the obfuscated samples we also looked into the basic challenges encountered during API monitoring approach. One problem with the previous call graph matching approach is that in a program many independent API calls are also issued. So if we reorder these independent API call sequences then it won't have any effect on the behavior of the program but will defeat the call graph matching approach. This drawback has been well described in the later sections. Another challenge is garbage API calls insertion. We could insert some irrelevant API calls which although won't have any effect on program behavior but will change the API call sequence.

Table 7: API call sequence used as signature for Bagle.A

1	GlobalAlloc	GetModuleFileName, CreateFile,UnmapViewOfFile GlobalFree
2	GetModuleFileName	
3	CreateFile	GetFileSize, CreateFileMapping, UnmapViewOfFile, CloseHandle
4	GetFileSize	UnmapViewOfFile
5	CreateFileMapping	MapViewOFile
6	MapViewOfFile	GlobalAlloc, UnmapViewOfFile
7	GlobalAlloc	lstrlen
8	lstrlen	
9	UnmapViewOfFile	
10	CloseHandle	
11	CloseHandle	
12	GlobalFree	

4.2 Data structures Used

A brief description of the data structures used in our proposed algorithm is given below.

1. We have used two arrays S_v and S_{ob} representing the traces of system calls of the signature and of any given process respectively. An example of a system call trace is shown as
"1:GlobalAlloc:40:2000:40:402770:158fa0: GetModuleFileName, CreateFile, UnmapVIewOfFile,GlobalFree" for S_v
and "868:GlobalAlloc:40:2000:40:402770:158cf0" for S_{ob}.

2. $MatchingList$ is an array which is used to store system call traces of any process similar to the APIs in the signature.

3. $SuspectedList$ is an array which stores interdependent API traces from $MatchingList$.

4. $Taint$ is an integer whose purpose is to store the number of independent APIs or APIs with no further dependency occurring in $SuspectedList$.

5. Ret is used to store the return value of an API.

6. $DependentAPI_i$ represents the APIs in the dependent array of API_i. For example Dependent[GlobalAlloc] will contain GetModuleFileName,CreateFile, UnmapVIewOfFile,GlobalFree.

7. $ARG[API]$ represents the arguments of API.

8. $RetValue(API[i])$ represents the return value of the specific $API[i]$.

9. $NumberofindependentAPIsin S_v$ is an integer representing the number of APIs in the signature with no elements in the *Dependent array*.

4.3 Proposed Algorithm

Now the proposed algorithmic model will take this signature comprising of some API calls along with their dependencies shown in table7 and a trace of API calls issued by any process as its inputs. The algorithm will first look for similar APIs between these two traces and then for dependencies among them. If it finds the same dependency in accordance with that among the APIs of the signature then it treats that process as malicious. The detection algorithm can be further divided into two sections as explained below.

4.3.1 The Initialization Part

This section initializes the necessary data structures and prepares for the actual detection routine. It takes two system call traces. One is the signature of the virus and the other is the system call trace of any process. It initializes a null array *MatchingList*. Its main purpose is for each API call in the signature it looks for the corresponding same API in the trace of the process. If a match is found then it pushes the corresponding API from the trace into *MatchingList*. It also initializes another array named *SuspectedList*. This array is only used to store the APIs from *MatchingList* with data dependency among them. Along with these two arrays another variable called *Taint* is also initialized to zero. Its main purpose is to keep track of the number of APIs with no child i.e. with no further dependencies which are being added to *SuspectedList*. To be brief this part of our algorithm can be treated as a filter part which takes out only the relevant API calls i.e. API calls common to both the traces and pushes them into *MatchingList*.

```
1: procedure The Initialization Part(S_v, S_ob)
2: Input:Signature of the virus, Trace of system
   calls of any process
3: Output: True or False, Depending on whether
   the trace is found or not
4:     SuspectedList ← ∅
5:     MatchingList ← ∅
6:     Taint ← 0
7:     for all API[i] ∈ S_v do
8:         for all API[j] ∈ S_ob do
9:             if (API[j]==API[i]) then
10:                Push this API[j] in MatchingList
11:            end if
12:        end for
13:    end for
14: end procedure
```

4.3.2 The Detection Part

After the initialization part, this part mainly builds the *SuspectedList* and checks for maliciousness. Here we have considered the first API in the signature of the virus as the *PrincipleAPI* as dependency between rest of the APIs start from here shown as the root of the call graph. So whenever an API similar to the first API of signature is found in the trace, it is added to *SuspectedList*. After this for every element in *SuspectedList*, this part checks whether that API has dependency or not. As per algorithm, $DependentAPI[i]$ refers to an array containing system calls having data dependency with $API[i]$ in the signature. If the dependent array is found non-empty then it stores the return value of that API. For each API in the array of $DependentAPI$, it checks for the corresponding API in *MatchingList*. If a matching is found, then it determines whether the dependency between the two APIs holds or not i.e. whether the return value of $API[i]$ is used as an argument in the corresponding dependent API. In the algorithm Arg[$API[k]$] represents the arguments of $API[k]$. If dependency holds then the corresponding dependent API from *MatchingList* is added to *SuspectedList*. This process is carried on for each and every element of *SuspectedList*. After the *SuspectedList* is built, the number of *independent APIs* present in the *SuspectedList* are calculated and for every matching, the value of *Taint* is increased by 1. Here for all the different occurrences of a single class of API, the value of *Taint* is increased only by one because *SuspectedList* can contain more than one API trace belonging to the same category under the same parent node. At last if the value of *Taint* is found equal to the number of *independent APIs* present in signature, then it is treated as malicious. Else the *SuspectedList* is cleared, the value *Taint* is reassigned to zero and the loop continues. After all the cases if the value of $Taint$ is still found to be zero, then the program is considered as benign. The algorithm is as shown below.

So considering the case of Bagle.a, after the initialization part *MatchingList* will contain the traces of system calls of S_{ob} with APIs as *GlobalAlloc*, *GetModuleFileName* etc., which are present in the signature. As per the signature,

the first API is *GlobalAlloc*. Therefore *GlobalAlloc* is considered as the principal API. Now in *MatchingList*, the detection part will look for *GlobalAlloc* and for every match, it will push that trace along with its arguments and return value into *SuspectedList*. Then *SuspectedList* is searched. Now at the *SuspectedList* generation part, among all the *GlobalAlloc*s in *MatchingList* which are being pushed into *SuspectedList* the first one is selected. As per the signature it can be seen that the dependent array of *GlobalAlloc* is not empty. So it stores its return value in *Ret* and for every API in the dependent array it looks for corresponding matching in *MatchingList*. Here the dependent array of *GlobalAlloc* contains the APIs *GetModuleFileName*, *CreateFile*, *UnmapViewOfFile*, *GlobalFree*. So at first it takes into account *GetModuleFileName* and searches for its match in *MatchingList*. If a match is found then it pushes this trace into *SuspectedList*. So after all the matches have been pushed, it does the same for the rest of the APIs in the dependent array. The above procedure is continued for all the elments in the *SuspectedList*. When the *SuspectedList* is built then as described in the above section, the number of *independent APIs* are calculated. If it coincides with the number of *independent APIs* present in the signature,then it is treated as malicious or else the value of *Taint* is reset and *SuspectedList* is cleared.

```
1: for all API ∈ MatchingList do
2:     if (API==First API of S_v) then
3:         Push API in SuspectedList
4:         for all API[i] ∈ SuspectedList do
5:             if (DependentAPI[i]!=0) then
6:                 Ret ← RetValue(API[i])
7:                 for all API[j] ∈ DependentAPI[i] do
8:                     for all API[k] ∈ MatchingList do
9:                         if (API[k]==API[j]) then
10:                            if (Ret==Arg[API[k]]) then
11:                                Push API[k] in SuspectedList
12:                            end if
13:                        end if
14:                    end for
15:                end for
16:            end if
17:        end for
18:        for all API[i] ∈ S_v do5
19:            if (DependentAPI[i]==0) then
20:                for all API[j] ∈ SuspectedList do
21:                    if (API[j]==API[i]) then
22:                        Taint ← Taint + 1
23:                        break
24:                    end if
25:                end for
26:            end if
27:        end for
28:        if (Taint==NumberofIndependentAPIsinS_v) then
29:            Trace of Virus is found
30:        else
31:            Taint ← 0
32:            Clear SuspectedList
33:        end if
34:    end if
35: end for
36: if (Taint==0) then
37:    The program is benign
38: end if
```

Now from the next element onwards *MatchingList* will be searched for the occurrences of *GlobalAlloc* and for each occurrence found, it will be pushed to *SuspectedList* and the above process will continue for each *GlobalAlloc* of *MatchingList*. At last as described in the algorithm the value of *Taint* is checked. if it is found to be zero then it will mean that for all the occurrences of *GlobalAlloc*, a *SuspectedList* was built but it did not contain the same number of *independent APIs* as per the signature. Then the algorithm exits with displaying the program as benign.

4.4 Challenges

As mentioned above, the main challenges of API call monitoring is the rearrangement of independent APIs and insertion of junk APIs. In the previous sections we described how independent API call reordering was unable to affect our algorithm. Now we will present a situation where some junk APIs will be inserted in the API instruction sequence of bagle.a. Let's assume that in an executable for a particular *GlobalAlloc* we found two *CreateFile*s in the trace where dependency holds as shown in the figure 2. Now suppose for the first *CreateFile* system call, dependency held with *GetFileSize* and *UnmapViewOfFile* while for the second *CreateFile*, dependency held with *CreateFileMapping* and *CloseHandle*. Now the problem can be realized when we compare it with the main call graph. We can observe that in the main call graph all four APIs *CreateFileMapping*, *CloseHandle*, *GetFileSize* and *UnmapViewOfFile* belong to the same parent node. But in the modified call graph the scenario is different. So in this case the the scanner should not consider the new executable as malicious. This drawback of graph matching based on API call monitoring is overcome by our algorithm. As explained above our algorithm will take every API belonging to *DependentAPI*[*j*] and will look for their match in the trace. So for each API under one parent node as per signature, their corresponding nodes in the trace will also have the same parent node. In a simple way we can say that our algorithm checks that every API call belonging to the dependent array of some system call must have the same parent node. The original call graph and the new API call graph showing this drawback are shown in the figures 1 and 2 respectively.

5. EXPERIMENTAL RESULTS

To determine the effectiveness of our algorithm we tess organized in the following manner. In section 2 some existing methods are described followed by highlighting some common obfuscation techniques used for deceiving antivirus software in section 3. In section 4 we proposed our scheme to detect obfuscated malwares. In section 5 we showed some of our experimental rested our algorithm against many obfuscated versions of our sample virus. We performed the testing in Windows XP operating system inside virtual box [5]. As said earlier we chose Bagle.A variant as our malware test case. Bagle was obtained from vx-heavens [6]. We used *Perl* for implementation purpose. The system call trace of different processes were obtained by using the tool *StraceNT*. To obtain the API calls of Bagle.A responsible for making a copy of itself we opened the virus in *OllyDbg* and referred to [16]. Though many obfuscators were available like Mistfall by z0mbie etc., but *OllyDbg* was used to insert code into the executable [19]. MSDN library was a source of great help in dealing with API calls [7]. We took a fresh copy of XP

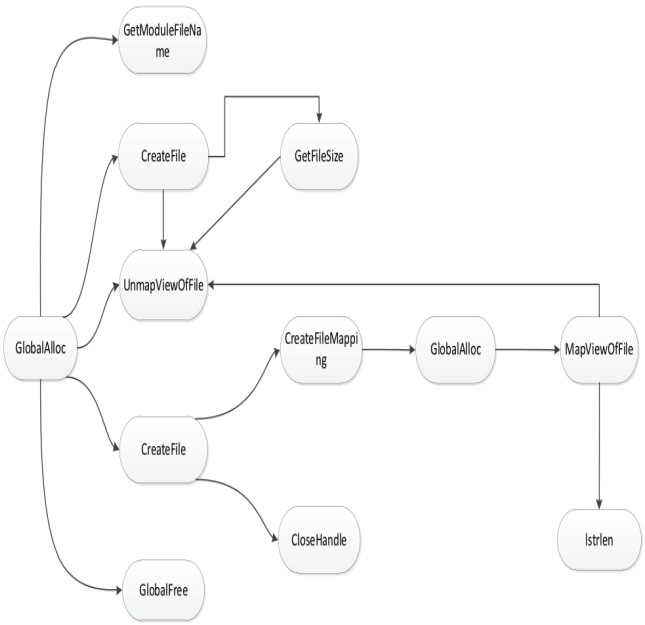

Figure 2: Call graph of Bagle.A with CreateFile inserted

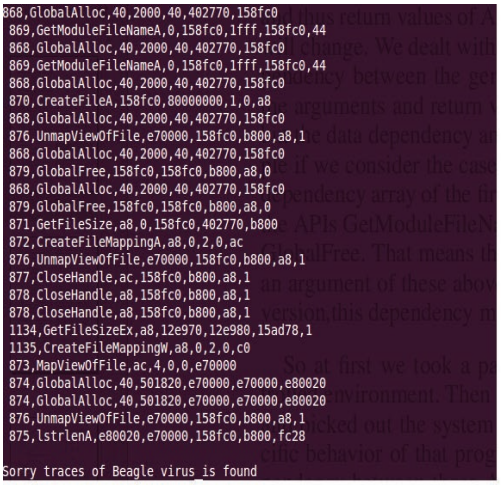

Figure 3: Screen shot of Obfuscated Bagle.A

without any security patches installed. For the first time we executed the virus in a virtual environment. During execution Bagle checks for system time to be earlier than 28 Jan, 2004. Also it checks for Internet connection using the API *INTERNETGETCONNECTEDSTATE*. So while debugging it in *OllyDbg* we changed the return value of those system calls for its safe execution. Now after execution we built the call graph and thus the signature. Now we tried to obfuscate the virus.

5.1 Malicious Code Testing

We prepared three categories of test cases. For each of the categories we prepared 10 obfuscated samples of the virus.

1. In the first one, we performed the simple obfuscation techniques described above. Here for every case we performed only one kind of obfuscation technique on the sample.

2. In the second category we tried to obfuscate the virus with a combination of multiple obfuscation techniques.

3. At last we used the garbage API insertion technique. In the previous example an obfuscated version with only *CreateFile* was shown. During experiments we inserted many different APIs and some common APIs from the signature also. We inserted those APIs between the sequence of APIs used as signature. We also made some cases in which the inserted APIs used the return values of previous APIs. The APIs were inserted carefully keeping in mind the semantics of the program.

Then we tested the original sample and the obfuscated versions of the sample with open source antivirus *ClamAV*. We found that ClamAV was not able detect the obfuscated samples. *ClamAV* was not even able to detect an obfuscated sample of bagle virus with only one NOP instruction inserted. Next we tested these obfuscated samples against our detection model. We found that for each obfuscated version in all the above three categories our model was able to successfully detect the virus. A screen shot displaying the result of obfuscated version of Bagle.A is shown in figure 3. Here the last argument of a system call entry represents its return value. It can also be observed that in some cases two occurrences of same trace are shown. This happened as two of the arguments of some API use the return value of a previous one. So in the algorithm as for each matching between return value and argument found, the trace is pushed into *SuspectedList*.

6. CONCLUSION AND FUTURE WORK

Though many antiviruse softwares have used different techniques to counter obfuscated malwares, still some of them are prone to this technique such as ClamAv. The advantage of virus writers is that they know the weak point of antivirus softwares i.e. byte sequence matching approach. This paper intends to provide some relief for antivirus researchers to tackle with obfuscation problem.

In this paper at first we described the case of malware obfuscation and showed through examples how this approach is being used by metamorphic and polymorphic viruses to defeat the antivirus softwares. We conducted testing against ClamAV antiivirus and our model. We also showed how our detection algorithm is sound against some common challenges faced by API call monitoring approach. Till now we have only applied the obfuscation techniques and detection routine on Bagle.A. For future work we will take different sample viruses and a set of benign programs to get a correct estimation of our false positive and false negative rate.

7. REFERENCES

[1] http://www.symantec.com/connect/blogs/w32stuxnet-installation-details.

[2] http://www.clamav.net/.

[3] http://www.datarescue.com/idabase/.

[4] http://www.ollydbg.de/.

[5] https://www.virtualbox.org/.

[6] http://vx.netlux.org/.

[7] http://msdn.microsoft.com/en-us/library/ms123401.aspx.

[8] D. Bruschi, L. Martignoni, and M. Monga. Detecting self-mutating malware using control flow graph matching. In *PROCEEDINGS OF THE CONFERENCE ON DETECTION OF INTRUSIONS AND MALWARE AND VULNERABILITY ASSESSMENT (DIMVA), IEEE COMPUTER SOCIETY*, pages 129–143, 2006.

[9] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.

[10] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 5–14, 2007.

[11] M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-aware malware detection. In *IN IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 32–46, 2005.

[12] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host.

[13] A. Lakhotia and E. U. Kumar. Abstract stack graph to detect obfuscated calls in binaries. In *In Proc. 4th. IEEE International Workshop on Source Code Analysis and Manipulation*, pages 17–26. IEEE Computer Society, 2004.

[14] J. Lee, K. Jeong, and H. Lee. Detecting metamorphic malwares using code graphs. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1970–1977, New York, NY, USA, 2010. ACM.

[15] L. O. M. Nicolas Falliere and E. Chien. W32.stuxnet dossier. Technical report, Symantec, 2011.

[16] K. Rozinov. Reverse code engineering: An in-depth analysis of the bagle virus. Technical report, Bell Labs, 2004.

[17] P. SzÃűr and P. Ferrie. Hunting for metamorphic. In *In Virus Bulletin Conference*, pages 123–144, 2001.

[18] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*, pages 297 –300, nov. 2010.

[19] z0mbie. http://z0mbie.host.sk.