***I.J. Wireless and Microwave Technologies*, 2011, 1, 13-22**
Published Online February 2011 in MECS (http://www.mecs-press.net)
DOI: 10.5815/ijwmt.2011.01.03

Available online at http://www.mecs-press.net/ijwmt

Detecting Polymorphic Buffer Overflow Exploits with a Static Analysis Approach

Guo Fan[a,*], Lu JiaXing[a], Yu Min[a]

[a] *College of Computer Information Engineering, Jiangxi Normal University, Nanchang, China*

Abstract

Remote exploit attacks are the most serious threats in network security area. Polymorphism is a kind of code-modifying technique used to evade detection. A novel approach using static analysis methods is proposed to discover the polymorphic exploit codes hiding in network data flows. The idea of abstract execution is firstly adopted to construct control flow graph, then both symbolic execution and taint analysis are used to detect exploit payloads, at last predefined length of NOOP instruction sequence is recognized to help detection. Experimental results show that the approach is capable of correctly distinguishing the exploit codes from regular network flows.

Index Terms: Exploit Code; Polymorphism; Abstract Execution; Symbolic Execution; NOOP Instruction Sequence

1. Introduction

In the world of Internet, hackers usually exploit remote vulnerabilities to gain the control of the hosts. The most serious threats are remote buffer overflow exploits. Well configured firewalls may prevent some vulnerable service from being attacked, but they can not stop remote exploits if the hosts have to provide these services for Internet.

Fig. 1 shows the typical structure of a buffer overflow exploit which includes three components:

1) The return address(RA) block, consecutively filled by the same 4 bytes, to cover the original function return address;

2) The payload, the real executable part, commonly a sequence of machine codes to generate a remote shell;

3) No operator sledge(NOOP), used to help RA locating the position of the payloads so that the payload is sure to be reached if RA points to any position in NOOP.

* Corresponding author:
E-mail address: guofan771210@yahoo.com.cn

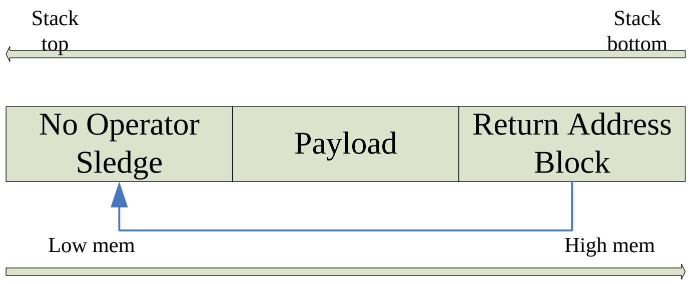


Fig. 1. Structure of a buffer overflow exploit

In the past, intrusion detection systems(IDS) can detect remote exploits by constructing a corresponding signature for every known exploit. But nowadays, exploits can evade the detection of an IDS by using polymorphism and self-modifying skills, and Metasploit[1], a famous exploit platform which can automatically generate thousands of variations for one exploit, provides many ways to encode the payload and NOOP of an exploit. It is impossible to construct the signatures of all the variations, so some new detection approaches have to be developed. The paper presents a novel method to detect polymorphic remote exploits.

2. Related Work

Dynamic detection and static detection are two major kinds of exploits detection methods. Dynamic methods usually using simulation environments(such as CPU simulator) to monitor the running state of polymorphic exploits, which are recovered to plain ones to really execute the payloads, and finally the plain exploits can be easily detected through pattern match of attach signatures by the IDS. The reference[2] proposed such a CPU simulator embedded in the IDS to detect polymorphic payloads. The key problem of a dynamic method is that it is incapable of covering all the paths and hackers may set up special conditions(such as a special day) to start the payloads, so the exploits may successfully evade detection since regular runs do not satisfy the conditions.

Static methods are more popular recently. In common with dynamic ones, they firstly have to distinguish executables instruction sequences from network flows, since regular network data do not include instructions. After the instruction sequences are recognized and disassembled, they using static analysis such as control flow analysis and data flow analysis to analyze whether the sequences include remote exploits.

Toth[3] proposed the concept of abstract execution used to disassemble instructions hidden in network flows. In his algorithm, the analysis focuses on the NOOP part and maximal executables length(MEL) is used to infer the existence of exploits. It was mainly used to detect plain exploits and expected a longer NOOP part, resulting in many false positives. Krugele[4] used the similarities of control flow structures to detect polymorphic worms, but it needed many groups of similar input data to make right conclusions. If hackers start the exploit only one time, it can not detect the remote exploit correctly. Chinchani[5] designed a fast detection approach that combined pattern match with data flow analysis and was based on the following ideas:

1) To gain a remote shell, the payload must include instructions that call well known system library functions, such as "int 80" under Linux or functions in Kernel32.dll under Windows, and these are obvious characteristics of an exploit;

2) A control flow graph(CFG) should be constructed as instructions were being disassembled. There are many invalid basic blocks in the CFG since much network data can not be disassembled into valid instructions, so the subsequent analysis only paid attention to valid blocks, and the time needed was reduced. Still[6] proposed a static detection architecture using both taint analysis and initialization analysis to discover polymorphic exploits. Via traditional pattern match methods, it firstly located an instruction sequence named GetPC to obtain the absolute address of current EIP. Then taint analysis used the address as a tainted seed and

propagated the tainted data until it reached a memory updating or written instruction whose operators were tainted by the address. Since GetPC was located by pattern match, it was difficult to recognize polymorphic and metamorphic GetPC sequences.

The paper proposes a novel static approach to detect polymorphic buffer overflow exploits and it is composed of 3 parts:

1) Inspired by the abstract execution[3] and the CFG construction method from reference[5], an algorithm is designed to almost linearly disassemble network data and construct a CFG at the same time;

2) In consideration of the characteristics of the popular polymorphic engines, symbolic execution[7] is combined with taint analysis to detect polymorphic payloads;

3) An efficient method is proposed to infer the existence of the NOOP part, reducing the false positive rate.

3. The Static Detection Approach

3.1. The CFG Construction Algorithm

The idea of abstraction execution is based on such a fact that Intel x86 architectures have variable length instruction sets and normal network data may be disassembled to valid instructions although most of them are actually not executable. A valid instruction means a correct instruction whose operators must be among reasonable ranges, that is to say, the operator can not access addresses exceeding the length of the network flow or forbidden by OS security access control. Abstract execution[3] means if the target address(TA) of a transfer instruction is statically analyzed, the next instruction being executed starts from TA immediately. Chinchani[5] marked each basic block with signs of Valid, Invalid, or Unknown, if the last instruction of the block is valid, invalid or the instruction is a transfer one and the target address is unknown.

The paper presents a CFG construction algorithm named LinearCFG visiting each byte of network flow only once to generate the CFG. The input of LinearCFG include B[1..n], the network flow with length n, startPos, the initial position in B, CBlock, current block pointer, BSet, the set of basic blocks, and ESet, the set of directed edges between blocks. Both BSet and ESet are dynamically changed as the algorithm runs interatively. The ideas of the algorithm mainly include:

1) The flag array visited ensures each byte is accessed once;

2) The abstract execution of the transfer instructions is realized by recursively calling the algorithm itself (label 2.2.2.1and 2.2.3.1);

3) Since GetPC sequences usually include “call” instructions, they are handled as conditional transfer instructions(label 2.2.3.1) and the block including “ret” instructions is marked as Unknown(label 2.2.2.4).

```
Name：LineraCFG
Input：B[1..n]，startPos，CBlock
InOut： BSet，ESet；
pos=startPos;
while (1) do {
1:   if pos>n，break；
2:  if visited[pos]==false{
2.1:   if InValid(pos,n,B) {   Visited[pos]=true;  pos=pos + 1;
             if (CBlock!=Null) {  CBlock.state=Invalid; CBlock=Null;    }
              continue;
       }//end Invalid
2.2:  else {  //f Valid(pos,n,B)
          Instr=GetInstr(pos,n,B)    visited[pos..pos+Instr.len-1]=true;
          If (CBlock == Null) {
              CBlock=Newblock() ; BSet=Bset∪{CBlock};
           }
           AppendBlock(CBlock,Instr);     pos=pos+Instr.len;
2.2.1:  if (Instr is not jump instruction)  continue;
2.2.2:  if (Instr is unconditional jump) {
2.2.2.1:      if( jump target is well-known) {
                  AddNewBlockToESet(BSet,ESet,CBlock,true);
                  LinearCFG(B,Instr.target,NBlock,BSet,ESet);
                  CBlock=Null; continue;
                 }
                 else { //jumpt target unknown
2.2.2.2:           CBlock.state=Unknown;  CBlock=Null; continue;
                 }
          }//end unconditional jump
2.2.3:   elseif (Instr is call or conditional jump){
2.2.3.1:    if ( jump target is well-known) {
                    AddNewBlockToESet(BSet,ESet,CBlock,true);
                   LinearCFG(B,Instr.target,NBlock,BSet,ESet);
                 }
  2.2.3.2:    else // jump target is unkown
                   CBlock.state=Unknown;
                      //false branch
                  NBlock=AddNewBlockToESet(BSet,ESet,CBlock,false);
                  CBlock=NBlock; continue;
                 }//end condtional jmp and call
2.2.2.4:     elseif (Instr is ret)  goto 2.2.2.2.
            } //end visited[pos]==false
3:     else { //visite[pos]=true
                NBlock=GetBlockFromPos(B,pos,BSet);
                If (NBlock == Null) goto 3.3; //invalid instr
3.1:              If (IsNotInstrHead(B,pos,NBlock,BSet)) {
                              pos=AddVisiteInstrToBlock(CBlock,
                                        B, pos,BSet);
                              goto 3.3
                   }
3.1.1:           If (pos==startpos) { //pos is jump target
                    If (IsFirstInstr(B,pos,NBlock,BSet)) {
                      Replace(ESet, CBlock, NBlock);
                      DeleteBlock(BSet,CBlock);
                    }
                    else
                      SplitBlock(BSet,ESet, NBlock,CBlock,B,pos);
                    break; //end the while cycle
                 }
3.2:            If (CBlock != Null) {
                   If (CBlock is empty) {
3.2.1:               Replace(ESet, CBlock, NBlock); DeleteBlock(BSet,CBlock);
                  }
                  else
                     ESet=ESet∪{(CBlock, NBlock,0)}
                }
3.3:            CBlock = Null;
                 pos=pos +1;  continue;
            }//end 3:
       }//end while
```

Fig. 2. Pseduo code of LinearCFG

4) The cycles, repeat function calls and the common target addresses may result that a byte is to be visited repeatedly. In these situations, when the algorithm is recursively called, the first instruction handled must have the condition visited[pos] = true and pos == startPos, as label 3.1.1 shows, then the pre-allocated blocks and edges are removed or some block need to be split to assure the first instruction of the block is the target of the transfer instructions.

5) The exceptions occur (label 3.1) when the target address is located not at the beginning but at the middle of a visited instruction. The handling procedure allocates a new block to contain all such instructions until an invalid instruction, an unvisited instruction, or the beginning of a visited instruction is reached. The block always contains few instructions since the different Intel x86 instruction sequences are quickly convergent[5].

The CFG constructed by LinearCFG is composed of many isolated sub CFGs each of which is a directed connected graph. An Invalid block must be the leaf node of a sub CFG and the last instruction of an Unknown block must be a transfer instruction with unknown target address.

Fig. 2 shows a polymorphic exploit code segment and the corresponding CFG generated from LinearCFG.

There are two transferring instructions call 0x9 and loopd 0xc, and the target address of call 0x9 is in the middle of the instruction itself, so that when the instruction at the address 0x9 is reached, the exception handling procedure at label 3.1 is started. The handling procedure ends when the instruction at address 0xb is reached. The left part of Fig. 3(b) is the result of LinearCFG after the algorithm is called recursively by the instruction call 0x9 and the instruction loopd 0xc is being reached. The right part is generated when the algorithm is again recursively called by the handling of the target address 0xc. Since the instruction xor at the address 0xc has been visited and it is not the first instruction of the belonged block B3, which is split into B31 and B32. When the recursively called algorithm for call 0x9 is completed, the instruction at the address 0xa is visited again, and the instruction is in the middle of inc eax, so the exceptional procedure starts again and generates blocks B4 and B5 until loopd 0xc is reached. The first instruction of B32 is not loopd 0xc so that B32 is split into the two blocks in the dot line rectangle in the right part of Fig. 3(b).

```
00:2BC9                     sub ecx,ecx
02:83E9B0                   sub ecx,-0x50
05:E8FFFFFFFF               call 0x9
09:FFC0                     inc eax
0b:5E                       pop esi
0c:81760EDC40D776           xor[esi+0xE],0x76D740DC
13:83EEFC                   sub esi,-0x4
16:E2F4                     loopd 0x0C
18:(Encrypted payload)
```

Fig. 3(a) A sample polymorphic exploit code segement

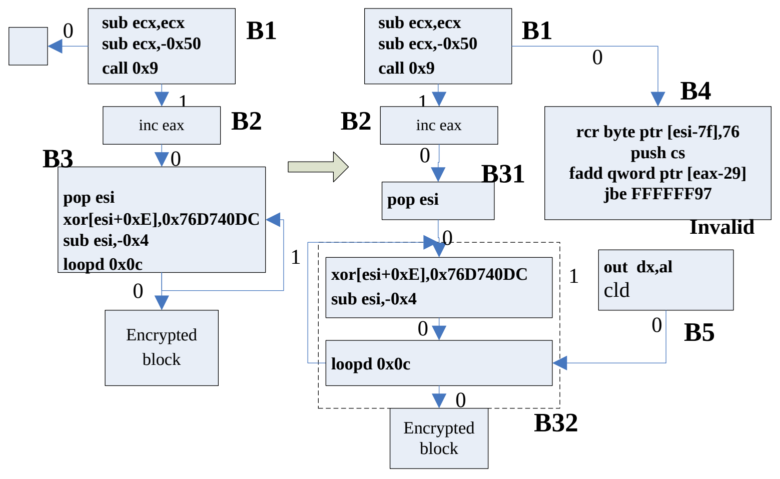


Fig. 3(b) The generated CFG for code segement in (a)

Fig. 3. An example of CFG constructed by LinearCFG

3.2. Polymorphic Payload Detection

As Fig. 3(a) shows, it is obvious a polymorphic payload has to obtain the absolute address of itself when the exploit is being run, so as to modify and transform the payload dynamically. The instruction sequence 0x05-0x0b denotes the procedure to get the address and the procedures is named GetPC by STILL[6] using pattern match to recognize it. If hackers transform GetPC sequences with metamorphism, it is very hard for STILL to correctly detect GetPC. The detection method presented in the section using symbolic execution to distinguish the transformed GetPC from normal instructions. Metamorphism means modifying a specified instruction sequence by inserting many meaningless instructions to change the pattern of the sequence so that it may evade detection. A meaningless instruction never updates or modifies the memory or the key registers.

From the entry node of each sub CFG, the method searches call instructions with known target address. When a call instruction is recognized, it starts symbolic execution from the target until one of the following conditions is satisfied:

1) Current instruction updates or modifies the memory;
2) Current instruction is a call or ret instruction;
3) Current instruction is a unconditional transfer instruction with an unknown target address;
4) The end of the sub CFG is reached;
5) The number of instructions symbolically executed exceeds the threshold;
6) The esp register is updated and the result can not be obtained symbolic computation.

The first condition means it is not a meaningless instruction; the second is used to avoid recursive symbolic execution which is unnecessary since every call instruction is to traversed; the third is obvious since symbolic execution could not find the next instruction; the fourth means the method reaching the end of the program slice; the fifth is for consideration of time performance and the threshold means the number of instructions a symbolic execution procedure may handle at most; the last means if the value of the esp register can not be recognized statically, the absolute address may not be obtained because the exploits have to get the address from the stack finally.

The symbolic execution procedure visits the same loop only once because the number of iterations of the loop can not influence the semantics of metamorphic GetPC sequence. For branch instructions, if the condition is unable to be determined statically, both branches are traversed by forking a new symbolic execution instance. Since metamorphic codes always contain few branch instructions, it hardly have any impact on the performance. If the current instruction is a pop instruction with a register operator like "pop ebx", it probably means a GetPC sequence is found out. At this moment, the symbolic execution procedure compares the address of the top of the stack with the address of the stack when the procedure starts initially, if the two addresses are equal, the sequence is a GetPC sequence and the procedure ends successfully.

When GetPC is recognized, the absolute address of the current instruction locates in the corresponding register. In order to transform the payload dynamically, a polymorphic exploit has to use the address directly or indirectly as the operators of memory updating instructions. Taint analysis combined with symbolic execution is used to assure whether the address is used to modify the instructions.

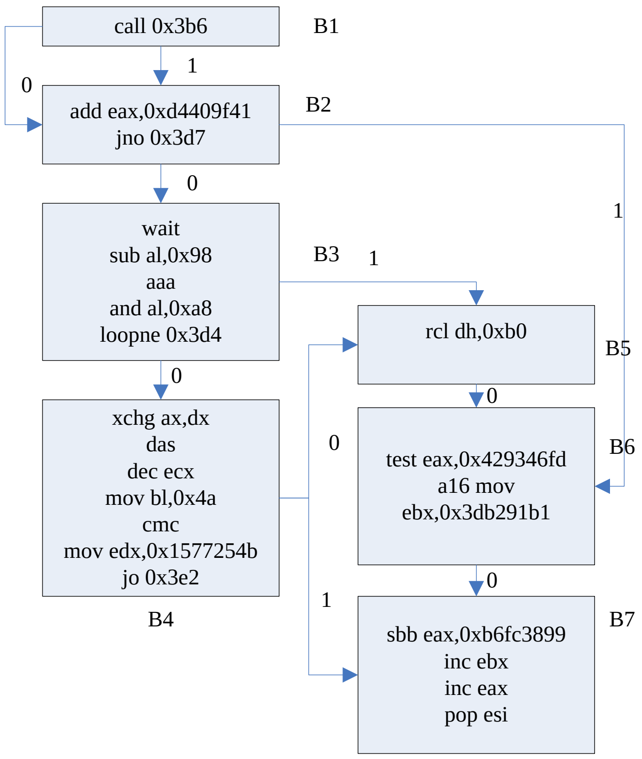
From the first instruction after the GetPC sequence, the register storing the absolute address is marked as tainted, then the taint states of the registers and memory locations are recorded as the symbolic execution proceeds, until one of the following conditions is satisfied: 1)the end of the sub CFG is reached; 2)the number of instructions symbolically executed exceeds the threshold; 3)the current instruction updates memory locations with tainted states.

Experiments show when GetPC is located and found out, the procedure usually reaches the instruction satisfying the third condition after symbolically handling only a few instructions. As Fig. 3(a) shows, when the absolute address is stored in register esi, esi is marked as tainted. It satisfies the third condition immediately since the next instruction "xor" uses the value of tainted esi as a part of the address of the operator, so that the instruction sequence is recognized as polymorphic exploits.

Fig. 4 is the metamorphic result of the GetPC sequence in Fig. 3(a) by the engine Opty2[8]. The corresponding CFG generated by LinearCFG is showed in Fig. 5.

```
3B1 E800000000            call 0x3b6
3B6 05419F40D4            add eax,0xd4409f41
3BB 711A                  jno 0x3d7
3BD 9B                    wait
3BE 2C98                  sub al,0x98
3C0 37                    aaa
3C1 24A8                  and al,0xa8
3C3 27                    daa
3C4 E00E                  loopne 0x3d4
3C6 6692                  xchg ax,dx
3C8 2F                    das
3C9 49                    dec ecx
3CA B34A                  mov bl,0x4a
3CC F5                    cmc
3CD BA4B257715            mov edx,0x1577254b
3D2 700E                  jo 0x3e2
3D4 C0D6B0                rcl dh,0xb0
3D7 A9FD469342            test eax,0x429346fd
3DC 67BBB191B23D          a16 mov ebx,0x3db291b1
3E2 1D9938FCB6            sbb eax,0xb6fc3899
3E7 43                    inc ebx
3E8 FFC0                  inc eax
3EA 5E                    pop esi
```

Fig. 4. An Eaxmple of Metamorphic GetPC



```
B1: call 0x3b6
B2: add eax,0xd4409f41
    jno 0x3d7
B3: wait
    sub al,0x98
    aaa
    and al,0xa8
    loopne 0x3d4
B4: xchg ax,dx
    das
    dec ecx
    mov bl,0x4a
    cmc
    mov edx,0x1577254b
    jo 0x3e2
B5: rcl dh,0xb0
B6: test eax,0x429346fd
    a16 mov
    ebx,0x3db291b1
B7: sbb eax,0xb6fc3899
    inc ebx
    inc eax
    pop esi
```

Fig. 5. CFG for the instruction sequence in Fig. 4

The symbolic execution starts from block B2, and since the condition of all the three branch instructions are not statically determined, all the eight paths are traversed. Interestingly, each instruction sequence by traveling a path in Fig. 5 is a GetPC sequence, like B1B2B6B7 and B1B2B3B4B5B6B7. In fact, the procedure end successfully when it reaches the instruction "pop esi" at the first time. Using pattern match as STILL[6] does may not recognize the GetPC sequence in Fig. 5.

3.3. NOOP Analysis

Since it is hard to exactly locate the address of the payload injected into the vulnerable program, exploits have to make use of NOOP Sledge(NS) depicted in Fig. 1 to assure the payload is successfully executed if the address falls in the range of NS. NS detection may help to reduce false positives of the method in section 2.2. NS is essentially meaningless instruction sequences which transform no operator instruction sequence into different ones. After a GetPC sequence is recognized, NS analysis obtains the position of the first call instruction and the corresponding block(EndBlock), furthermore, the entry node(Entry) of the sub CFG containing the GetPC sequence is also located. All the possible paths from Entry to EndBlock are traversed to find out a consecutive meaningless instruction sequence whose length is more than the predefined threshold. The sequence is inferred as an NOOP sledge.

```
NSAnalysis(BSet,ESet,CurB,PL,PrevL) {
   if (CurB.L3 > PL)  return CurB.L3;
   if (CurB.L1 + PrevL > PL)
      return CurB.L1+PrevL;
 if (CurB.L4 != 0) {
      PrevL = PrevL + CurB.L4;
    if (PrevL > PL) return PrevL;
 }
   else
      PrevL=CurB.L2
   for each (CurB, NextB) in ESet {
      L = NSAnalysis(BSet,ESet,NextB,
               PL,PrevL);
        if (L > PL) return L;
 }
 return 0;
}
```

Fig. 6. Step 3 of NS analysis

The analysis is composed of the following procedures:

1) The sub CFG is transformed to CFG', for each block B in CFG', there must be a path (B, B1) (B1, B2) ... (Bn, EndBlock), and each pair of (Bi, Bj) belongs to ESet. The last instruction of EndBlock in CFG' is the call instruction of GetPC.

2) Computing the sequence length for four kinds of meaningless instruction sequence in each block of CFG'. They are the sequence include the first instruction(L1), the sequence include the last instruction(L2), the maximal consecutive instruction sequence(L3), and the sequence contains all the instructions(L4).

3) According to the threshold PL, each block is handled like Fig. 6, the length of instruction sequence is accumulated between neighbor blocks until the length exceeds PL or the end of CFG' is reached. PrevL means the remaining threshold when the analysis reaches the current block.

4. Experiments Evaluation

An prototype is implemented under VC 6.0 platform and LinearCFG implementation is on the basis of the program from Toth[3]. The experiments contain two parts. Part one is using the prototype to detect the polymorphic payloads provided by Metasploit[1] platform in order to evaluate the correctness of the approach. Part two is to evaluate the false positive rate by applying the prototype to normal network flow and regular applications.

Ten typical payloads are selected and are transformed by five polymorphic engines, x86/call4_dword_ptr(1), x86/countdown(2), x86/jmp_call_additive(3), x86/nonalpha(4) and x86/nonupper(5). Each polymorphic payload is combined with 20 different NOOP sledges whose length is between 50 and 250. The number of test polymorphic payloads is 1000. The number of instructions to be symbolic executed is limited to not more than 100, and the number to be NS analyzed is set to 20. Table 1 shows the experimental results, that is, each polymorphic payload is correctly detected.

Table 1. Detection of polymorphic payloads

	1	2	3	4	5
linux/x86/shell/bind_tcp	x	x	x	x	x
linux/x86/shell/reverse_tcp	x	x	x	x	x
linux/x86/exec	x	x	x	x	x
windows/exec	x	x	x	x	x
windows/dllinject/bind_tcp	x	x	x	x	x
windows/dllinject/reverse_tcp	x	x	x	x	x
windows/shell_bind_tcp	x	x	x	x	x
windows/shell_reverse_tcp	x	x	x	x	x
windows/shell/bind_tcp	x	x	x	x	x
windows/shell/reverse_tcp	x	x	x	x	x

To collect normal network data, the gateway of a LAN is sniffed to record all HTTP requests and replies for 24 hours and 12186 packets are finally obtained. The number of false positives is zero after the prototype is applied. The disassemble result of sample packets are manual analyzed, and the results show call instructions rarely appear in the disassembled instructions from normal network data, if such a call instruction is found occasionally, it is hard to satisfy the conditions of GetPC. The applications in directory /bin of Redhat Linux 9.1 are also analyzed by the prototype. Also, there are no alerts for these applications, since regular applications never use GetPC.

Although the experiment results make a good conclusion, the approach presented here has some limitations.

1) GetPC detected by the approach is base on the call instruction, while not all polymorphic engines are based on call instructions. For instance, some engines are based on float instructions like "fsetenv";

2) The number of instruction to be symbolically executed is limited, so that hackers may use a longer GetPC sequence to evade detection;

3) If the call instruction with known target address is replaced by a indirect call sequence, such as "mov eax 0x6, mov ebx, eax, call eax", the approach fails;

4) There may be a false positive if the instruction sequence like 0x5-0x13 in Fig. 3(a) is embedded into normal network flows.

The experiment results demonstrate the better effect of the aggregation algorithm, and the results may be much better if the configuration of weights and expected similarity are tuned.

5. Conclusion

The paper proposes a novel static analysis approach to detect polymorphic buffer overflow exploits on the basis of past researches. The idea of abstract execution is adopted to construct the CFG when the network data is being disassembled. Symbolic execution is used to detect GetPC sequence of a polymorphic exploit, and taint analysis is combined to assure the existence of the exploit. NS analysis may help reducing the false positive rate of the approach. The experiments on polymorphic payloads, normal network flow and regular applications show that the approach is efficient to detect polymorphic buffer overflow exploits.

References

[1] H.D. Moore. The metasploit framework[EB/OL]. http://www.metasploit.com, 2010.
[2] M. Polychronakis, K.G. Anagnostakis. Network-level polymorphic shellcode detection using emulation[C]. In Proceedings of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment, 2006
[3] T. Toth, C. Kruegel. Accurate buffer overflow detection via abstract payload execution[C]. Recent Advance in Intrusion Detection, 2002
[4] C. Cruegel, E. Kirda. Polymorphic worm detection using structural information of executables[C]. Recent advance in Intrusion Detection, 2005
[5] R. Chinchani R, E. Berg. A fast static analysis approach to detect exploit code inside network flows[C]. Recent advance in Intrusion Detection, 2005
[6] X.R. Wang, Y.C. Jhi, S.C Zhu, P. Liu. STILL: Exploit code detection via static taint and initialization analyses[C]. In Annual Computer Security Applications Conference, 2008
[7] J.C. King. Symbolic Execution and Program Testing[J]. Communications of the ACM, 19(7):385-394, 1976.