J. Software Engineering & Applications, 2010, 3, 965-971
doi:10.4236/jsea.2010.310113 Published Online October 2010 (http://www.SciRP.org/journal/jsea)

Program Slicing Based Buffer Overflow Detection*

Yingzhou Zhang[1,2,3], Wei Fu[1,2], Xiaofei Qian[1], Wei Chen[1]

[1]College of Computer, Nanjing University of Posts and Telecommunications, Nanjing, China; [2]State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China; [3]State key Laboratory Of Networking & Switching Technology, Beijing University of Posts & Telecomm., Beijing, China.
Email: zhangyz@njupt.edu.cn

Received July 26th, 2010; revised August 16th, 2010; accepted 26th, 2010.

ABSTRACT

The development of the information technology has brought threats to human society when it has influenced seriously the global politics, economics and military etc. But among the security of information system, buffer overrun vulnerability is undoubtedly one of the most important and common vulnerabilities. This paper describes a new technology, named program slicing, to detect the buffer overflow leak in security-critical C code. First, we use slicing technology to analyze the variables which may be with vulnerability and extract the expressions which will bring memory overflow. Secondly, we utilize debug technology to get the size of memory applied by the variable and the size of memory used for these code segments (*the slicing result*) *further. Therefore we can judge whether it will overflow according to the analysis above. According to the unique excellence of program slicing performing in the large-scale program's debugging, the method to detect buffer overrun vulnerability described in this paper will reduce the workload greatly and locate the code sentences affected by corresponding variable set quickly, particularly including the potential vulnerability caused by parameter dependence among the subroutines.*

Keywords*: *Program Slicing, Buffer Overflow, Inter-Procedure Slicing, Debug, System Dependence Graph

1. Introduction and Related Work

As early as the beginning of 1970s, buffer overflow has been wildly believed that it is caused by the defects of C language's design model. Array and pointer references are not automatically bounds-checked, so programmers must be up to do these checks by themselves. It is more important that many of the string operations supported by the standard C library such as strcpy(), strcat(), sprintf(), gets(), are unsafe actually. They directly copy data with unknown size to the fixed buffer (as shown in **Figure 1**), causing data overwriting in memory, access violation, or execution of malicious code designed by hackers. The data from CERT shows that 55 percent of general injection attacks are buffer overflow attacks [1], and that among the thirty new vulnerabilities happened during April 19 to May 30 in this year, buffer overflow holds 40 percent [2].

Currently, there are some problems to be solved. Tools with static buffer overflow detection based on string matching algorithms maintain a high rate of false alarms. If the functions existed in the program match the vulnerability database carrying by themselves, they give the corresponding reports.

Now, the common detection methods check the bound of arrays. They regard the memory space variable applied as integer range, for example, char e.g. [10], and its range is [1,10]. When data are copied into it, we must judge whether the buffer overflows. But problems are still existed. First, we are hard to know how many sizes of the buffer have been used. Because we used to call library function to operate the string, and almost all of the library

```
// no buffer overflow
void NoVulFunc(void)
{
char dest[6];
……
strcpy(dest,"Hello");
……
}
```

```
// buffer overflow
void VulFunc(void)
{
char dest[6];
……
strcpy(dest,"HelloWorld");
……
}
```

Figure 1. An example of a buffer overflow.

*This work was supported in part by the Natural Science Foundation of China (60703086, 60873231, 60873049, 60973046), Jiangsu Natural Science Foundation of China (BK2009426).

function's source code is private or existed in DLLs. Second, these tools are hard to deal with a program with multiple procedures. Lastly, tracking every variable's buffer is almost impossible in a large project.

To solve these problems, many researchers have presented methods and tools for detecting buffer overflows.

1) *The static detection methods based on string matching*. Through string matching, the UNIX's tool, *grep*, can be used to find the unsafe library function call. The tool *RATS* [3,4], developed by Secure Software inc., also studies the unsafe function call in the source code. It matches with the vulnerability database, and then gives the description of the vulnerabilities. Those tools as mentioned above can hardly analyze the semantics and grammar of a program, thus they have many limitations and high false positive.

2) *The detection method based constraint analysis. Splint* [5], developed by university of Virginia, is a static analysis tool used to detect the vulnerabilities in the C program. It adds information of constraint to the source code, and then makes lexical, grammar and semantic analysis for the source. It judges the leaks that may happen in a program and gives the related instructions. From the workflow of the Splint, its limitation is that it requests analysts to be known the source well, and to add notations to the interesting variables and functions.

3) *The dynamic detection methods*. The dynamic detection tool of buffer overflows first inserts some detecting codes in the place where it may happen buffer overflow, then compiles and executes the codes after inserting, and finally, detects whether overflow happens during the execution. But the dynamic detection has defects of efficiency and coverage.

Against the problems described above, this paper proposes a novel method of program analysis, program slicing [6-8], to detect the buffer overrun.

2. Program Slicing

Program slicing [6], originally introduced by Weiser, has been widely used in maintenance of software, program debugging, software testing, code analysis, reverse engineering and so on. For example, during the program debugging, we hope to allocate the codes causing the error. But perhaps the program is too large to find by our hands. If we apply program slicing, we can exclude the codes that have nothing to do with the error, then allocate the error in a smaller range.

In general, program slicing technology has experience a lot: from static slicing to dynamic, from forward slicing to backward, from single procedure to multiple, from non-distributed slicing to distributed, etc.

The slice of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p (see **Figure 2**). This concept, originally discussed by Mark Weiser, can be used to isolate individual computation threads within a program. In Weiser's terminology, a slicing criterion is a pair <p,v>, where p is a program point and v is a subset of the program's variable.

With the expansion of the program scale, it is inevitable that program contains multiple procedures. So it appears more important to study inter-procedural slicing. Regarding inter-procedural slicing as a question of graph's reachability, S.Horwitz *et al.* [8] introduce system dependence graph (SDG) to represent the program's dependence graph (PDG).

PDG is a directed graph connected by different kinds of vertexes and some edges (e.g. **Figure 3**). And the vertexes include function's entrance node, declaration node, assignation node, control predicate node, function's call site node, parameter node, the FINAL_USE node of every variable, etc. The edges include control dependence edge of program's circuit, parameter-in and parameter edges generated by call and other data dependence edges. If the definition of variable x in node n is a reachable definition of node m, node m is *data dependent* on n. Control dependents only exist in condition expression and inside the loop expression.

SDG (e.g. **Figure 4**) composes of procedure dependence graph, which is connected by edges that represent direct dependences between a call site and the called procedure and edges that represent transitive dependences due to calls.

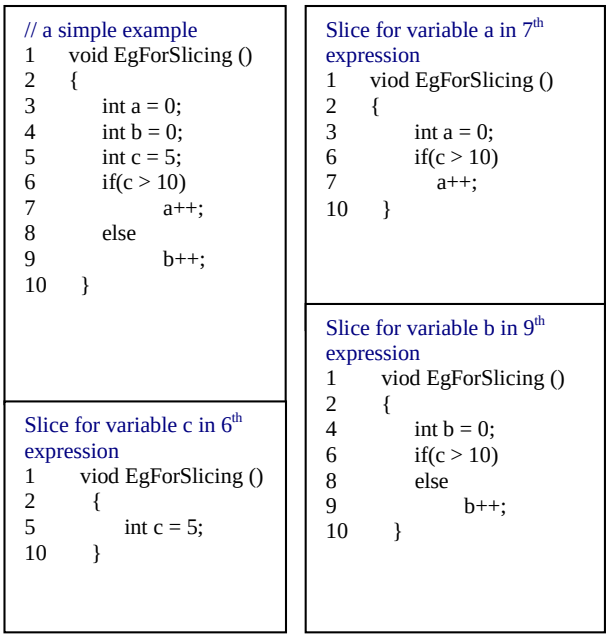

Figure 2. A sample of program slicing.

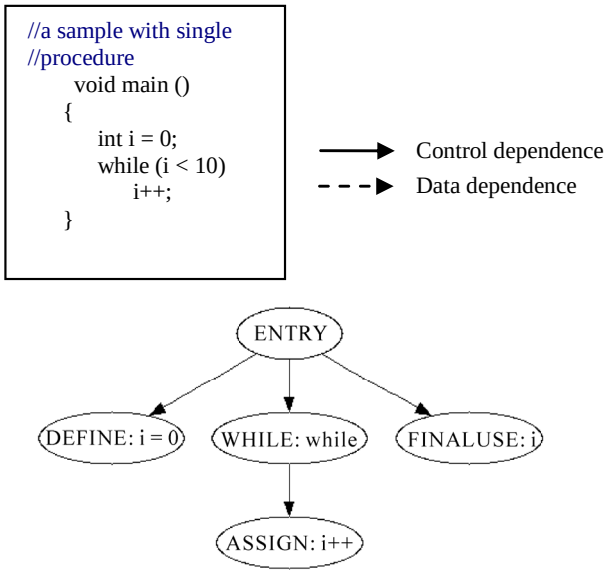

Figure 3. A program and its PDG.

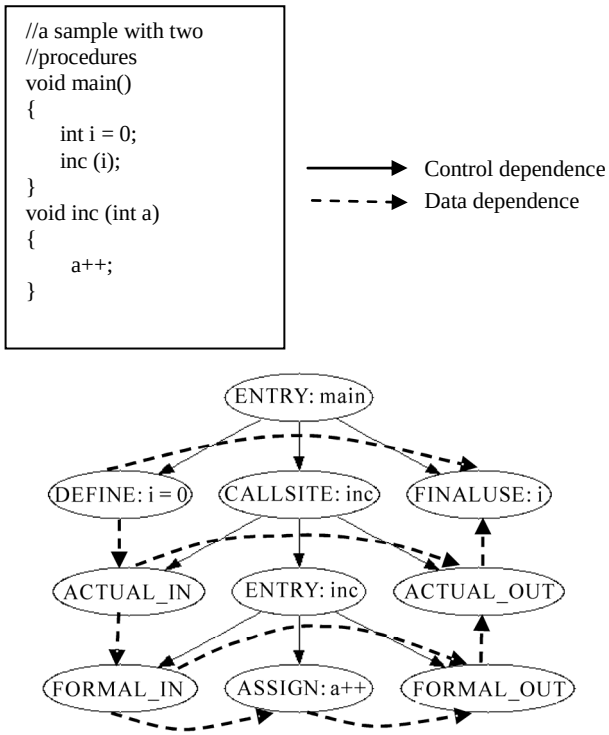

Figure 4. A sample program and its SDG.

For every call site, we use four sorts of vertices to denote the parameter passing: on the calling side, information transfer is represented by a set of vertices called actual-in and actual-out vertices. These vertices, which are control dependent on the call-site vertex (see **Figure 4**), represent assignment statements that copy the values of the actual parameters to the call temporaries and from the return temporaries, respectively. Similarly, information transfer in the called procedure is represented by a set of vertices called formal-in and formal-out vertices. These vertices, which are control dependent on the procedure's entry vertex, represent assignment statements that copy the values of the formal parameters from the call temporaries and to the return temporaries, respectively.

3. An Algorithm of Buffer Overflow Detection

In this section, we will show in detail an algorithm of detecting buffer overflow through program slicing. The algorithm includes four steps as follows.

Step 1. *Constructing PDGs*

Through the lexical and syntax analysis (with Flex and BYACC) of the program to be detected, we first construct the vertices in PDGs. There are usually seven kinds of vertexes as follows.

1) The beginning vertex of compound statements which includes function, if-else, switch statements and so on;

2) The end vertex of compound statements;

3) Call-site vertex;

4) Actual-in and actual-out vertex;

5) Formal-in and formal-out vertex;

6) FinalUse vertex;

7) Others vertex such as ordinary definition statements (e. g.: *int* i), predicate vertexes (e. g.: the judgment part of a *if* statement) and jump statements (e. g.: *break*, *goto* and so on).

Then we construct the edges of PDGs by analyzing the program dependences (includes control dependences and data dependences, see **Figure 4**).

Step 2. *Constructing SDG*

According to the PDGs in Step 1, the SDG can be constructed by the following substeps:

1) For each call site, a call edge from the call-site vertex to the corresponding procedure-entry vertex, is added into the PDG related.

2) For each actual-in vertex v at a call site, we add in PDGs a parameter-out edge from v to the corresponding formal-in vertex in the call procedure;

3) For each actual-out vertex v at a call site, we add in PDGs a parameter-out edge to v from the corresponding formal-out vertex in the called procedure;

4) We finally add in PDGs an edge between actual-out vertex to actual-in vertex if they are reachable between the corresponding formal-out and the corresponding formal-in.

Step 3. *Computing program slices by traversing SDG in two phases*

Supposing that we want to computer the inter-procedure

slice of the variables in vertex n.

Phase 1: Starting from the vertex n, we travel reversely the SDG through the control dependence edges and data dependence edges (not including parameter-in edges), and then mark all of reachable vertexes.

Phase 2: Starting from the vertexes marked in phase 1, through the control edges (not including call-site edges) and data dependence edges (not including parameter-out edges), we mark all of reachable vertexes.

The result of the inter-procedure slice is the union set of the marked vertexes in above two phases.

Step 4. *Detecting Buffer Overflow*

For each variable needed to analyze, we can use a data structure to store the information of usage. The data structure includes variables, the flags of overflow, the usage information and the total size of memory applied by a variable. For example, char p [10], the detection model about p is as follows (see **Figure 5**).

From the abstract syntax tree obtained by lexical analysis and syntax analysis, we can get each call function. Through the inter-procedure slicing of the parameters of the vulnerable functions, we then can obtain a piece of executable code segment which may affect these parameters. At last, we watch the memory of the variable for judging whether the buffer related is overflow, by setting breakpoints in a debugger at the beginning of the main function and the vulnerable function.

4. Implementation

Through the intermediate representation AST of a program (from the lexical and syntax analysis of the program), we can construct some useful graphs such as PDGs and SDGs. In PDGs, the data structure of nodes and edges are as follows.

```
struct PDGNode {
        CFGType type;
//type of node, such as "DEFINE", "IF", "WHILE". etc
        StructId    astId;
      //node's ID in AST
        PDGId       father;
      //connect to the control predicate
        PDGId       tBranch, fBranch;
      //the PDGID of true branch or false branch
            PDGId       ifd;
      //post-dominator
        ListId      prevHead;
    //the first node'ID of the father node table
        PDGId       callLink;
    //link to the first function call site
        DepEdgeId cdFirst;
    //the ID of the node's first control dependence edge
        DepEdgeId ddFirst;
    //the ID of the node's first data dependence edge
```

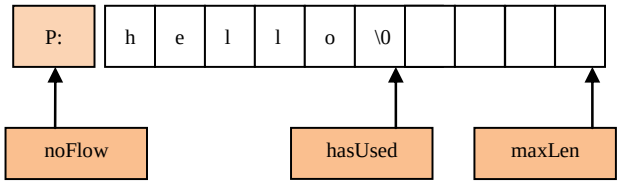

Figure 5. The detection model of buffer overflow.

```
        };
struct DepEdge {
            int         depSrc;
  //the PDG ID of dependence source
            int         depDest;
  //the PDG ID of dependence destination
            DepEdgeId srcNext;
//the next dependence edge with the same dependence
  //source
            DepEdgeId destNext;
//the next dependence edge with the same dependence
  //destination
        };
```

According to the algorithm in above section, we can construct SDGs from PDGs by adding some edges with the following functions.

1) Adding control dependence edges from call-site to the callee's body, through the function AddEdge (CTRL_EDGE, callSite, entry).

2) Adding data dependence edges from actual-in to formal-in, through the function AddEdge (DATA_EDGE, Actual_in, Formal_in)

3) Adding data dependence edges from formal-out to actual-out, through the function AddEdge (DATA_EDGE, Formal_out, Actual_out)

The function AddEdge inserts dependence edges to the dependence table according to the edge's style. Then we assign an ID of the new edge to the destNext (a member variable of DepEdge). The following codes show in detail the implementation of AddEdge.

```
AddEdge (int kind, int from, int to)
  {
      int lastEdge;
      DepEdge dep (from, to);
      depTable. insert (dep);
      switch (kind)
      {
        case CTRL_EDGE:
          lastEdge = pdgTable [from]. cdFirst;
          break;
        case DATA_EDGE:
          lastEdge = pdgTable [from] .ddFirst;
          break;
        }
      if (lastEdge < 1)
```



```
    //if this node has no dependence edge
    {
    pdgTable [from]. cdFirst = depTable. getCurId();
    return;
    }
  //if the node has dependence edge already
      while ((depTable [lastEdge]. depDest! = to)&&
         depTable [lastEdge]. destNext >= 1)
       lastEdge = depTable [lastEdge]. destNext;
       depTable [lastEdge]. destNext = depTable.getCurId ();
}
```

The dependence between the actual parameter can be formed according to the dependence among the corresponding formal parameters. When we analyze the library function, we suppose that each actual-out is dependent on the actual-in. So we need not go deep into the inner of the library function body. The detail codes of the dependence among formal parameters are as follows.

```
void BuildInterActualParameterDep (int fnId)
{
  //paramNum is the number of the parameter
    for(int i = 0; i < fnTable[fnId]. paramNum; i++)
  //judge whether the corresponding formal-in can
  //reach the corresponding formal-out
   if (IsReachable (formal-in, formal-out))
   //Add edge between the corresponding actual-in
  //and the corresponding actual-out;
    AddEdge (DATA_EDGE, actual-in, actual-out);
}
```

According to the algorithm described in section 3, we need to traverse the SDG in two phases. Because parameter-out edges are not followed, the traversal in phase 1 does not descend into procedures. But the effects of such procedure are not ignored. The presence of transitive flow dependence edges from actual-in to actual-out vertices permits the discovery of vertices that can reach the vertex you want to slice through a procedure call, although the graph traversal does not actually descend into the called procedure. In phase 2, because call edges and parameter in edges are not followed, the traversal does not ascend into the calling procedure; the transitive flow dependence edges from actual-in to actual-out vertices make such ascents unnecessary. So we can solve the call-context problem (as shown in the following codes).

```
void InterProSlice (SDGs, PDGId vulPdgNode)
 {
  //phase 1: traverse in calling procedure
  ReachingNode (s, vulPdgNode, {parameter_out});
  //phase 2: traverse in called procedure
  //vSet is a set marked in phase 1;
  ReachingNode(s, vSet, {parameter_in, call});
  }
void ReachingNode(SDG s, vSet, kind)
        //vSet is the set of vertex
        //kind is type of the vertex
{
  stack<PDGId> nodeStack;
  push the vertices that exist in the vSet to the stack;
  while (!nodeStack .IsEmpty())
   {
      pop a vertex v from the stack;
      mark the vertex v;
      while (if v's depTable is not empty)
      {
     push the vertices existed in the depTable to the stack;
      }
    }
}
```

After obtaining the result of the slicing, we set breakpoint at the beginning of the main function and at the place of the vulnerable function, then call the debugger (for example: the debugger embedded in Microsoft Visual Studio or the Zeta debugger [9]) to execute by step until the end. The implementation is as follows:

```
void BufferOverDetect (int start, int end, int vulPos, char * fileName)
        {
            char buf [10];
            ZD_LoadProgram (fileName);
            ZD_SetBreakPoint (start,true);
            ZD_SetBreakPoint (vulPos,true);
            ZD_RunTo (start);
            While (start <= end)
        {
            ZD_RunTo (++start);
        }
      /*the function below is used to read a byte of
        content starting from the memory address of
        add to the buf. The add is passed as follows: if
        you want to check whether vul (vul is defined
        like this: char vul [10]) will be overflowed, we
        just need to watch the content of the vul [10],
        and the add is & vul [10].*/
        ZD_Read (add, 1, buf);
  }
```

5. A Sample

In this section, we will show based on program slicing a sample (see **Figure 6**), where the variable of vulBuf will be overflowed.

Due to the restriction of space, parts of the vertices in **Figure 6** have been abbreviated shown as follows.

```
D:noR      = DEFINE:noRelated;
D:noRBuf = DEFINE:noRelatedBu;
D:vulBuf = DEFINE:vulBuf;
A:noR++ = ASSIGN:noRelated++;
```


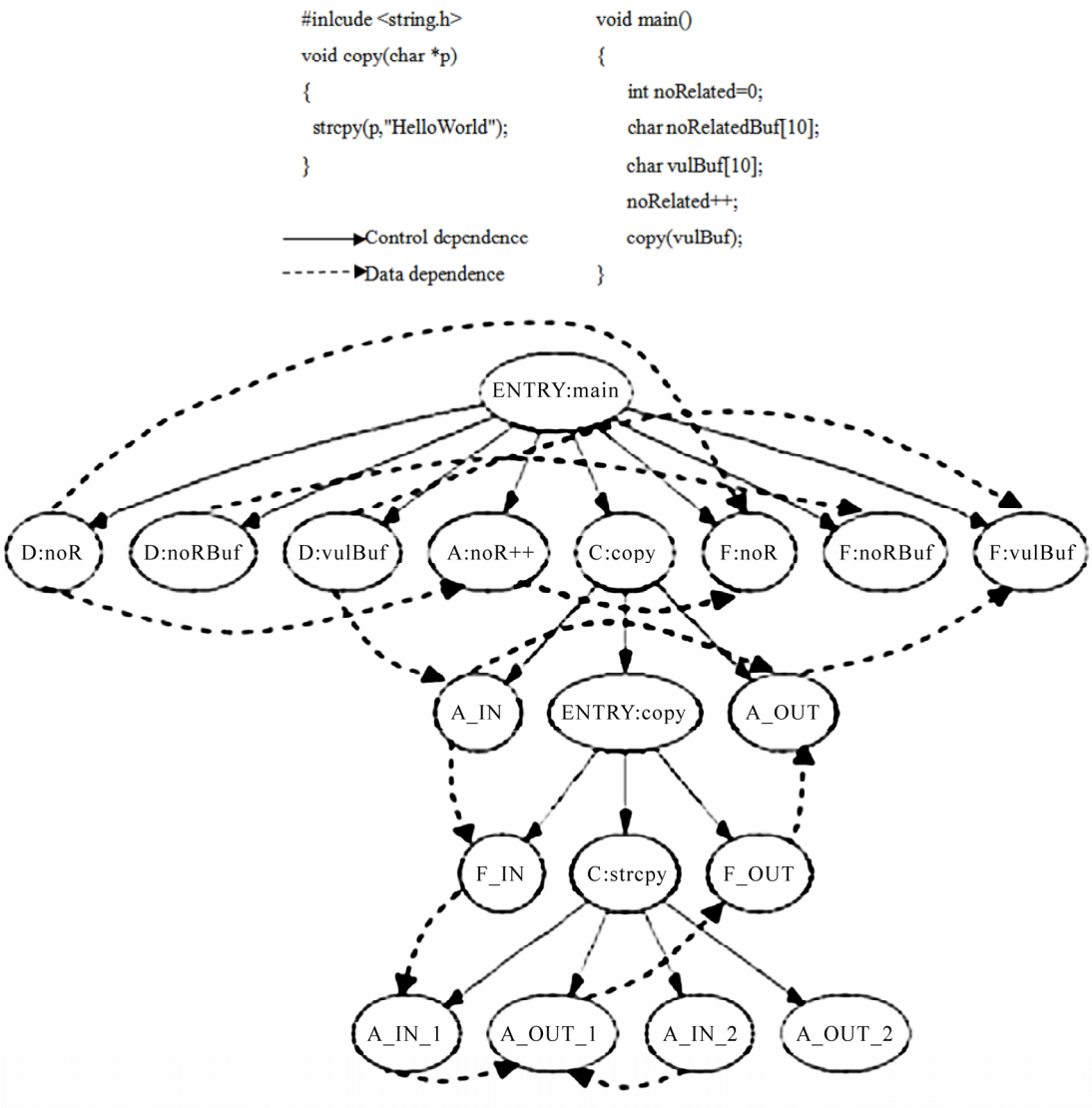

Figure 6. A sample of buffer overflow and its SDG.

C:*copy* = *CALL_SITE*:*copy*;
F:*noR* = *FINALUSE*:*noRelated*;
F:*noRBuf* = *FINALUSE*:*noRelatedBuf*;
F:*vulBuf* = *FINALUSE*:*vulBuf*;
A_IN = *ACTUAL_IN*;
A_OUT = *ACTUAL_OUT*;
F_IN = *FORMAL_IN*;
F_OUT = *FORMAL_OUT*;
*A_IN*_1 = *ACTUAL_IN*_1;
*A_OUT*_1 = *ACTUAL_OUT*_1;
*A_IN*_2 = *ACTUAL_IN*_2;
*A_OUT*_2 = *ACTUAL_OUT*_2;

After constructing the SDG, we start to find the vulnerable function call with the matching. Then we can find the strcpy in the copy may cause overflow. So we make inter-procedure slicing for variable p, and obtain the result showed in **Figure 7**.

Then we call the debugger, and set breakpoint at the beginning of the main function and at the place of the strcpy, by starting to execute by step. At last, we will find vulBuf [10] equals character of NULL (see **Figure 8**). This shows that vulBuf has been overflowed.

After debugging, we will give the corresponding report.

There are some advantages in our detection tool based on program slicing. First, this tool improves the accuracy greatly compared with the ITS4 and RATS which use string matching to detect the buffer overflows. Second, through program slicing, we can get rid of the useless codes. Compared with the detecting tool that sets a constraint for each variable and watches its value, our tool can reduce the variables needed to watch, improve the


```
# include <string.h>
void copy(char *p)
{
        strcpy (p,"HelloWorld");
}
void main()
{
       char vulBuf [10];
       copy (vulBuf);
}
```

Figure 7. The slice result of the sample in Figure 6.

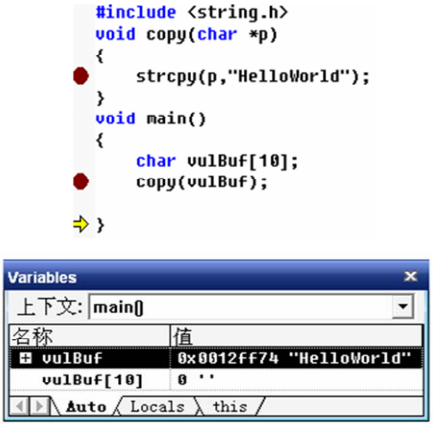

Figure 8. The result of a detection debugger.

performance. Third, compared with the Splint, our tool needs not to insert information of constraint to the source. So the analyst needs not learn the program a lot.

6. Conclusions

This paper introduces inter-procedure slicing to solve the problem of detecting buffer overflow. Compared with the other methods, this method has the high performance and excellent precision. But our tool is only fit for the programs coded by C, and it is still blindness in the object-oriented languages. So eliminating the blindness will be our further work.

REFERENCES

[1] "CERT/CC.Vulnerability Notes by Metric," http://www.kb.cert.org/vuls/bymetric?open&start=1&count=20

[2] "US-CERT Recently Published Vulnerability Notes," http://www.kb.cert.org/vuls/atomfeed?OpenView&start=1&count=30

[3] "Secure Software, Rough Auditing Tool for Security, (RATS)," Secure Software Inc, http://www.secure software.com

[4] J. Viega, J. T. Bloch, T. Kohno and G. McGraw, "ITS4: A Static Vulnerability Scanner for c and c++ Code," *Annual Computer Security Applications Conference*, Hawaii, 2000, pp. 257-267.

[5] D. Evans and D. Larochelle, "Improving Security Using Extensible Lightweight Static Analysis," *IEEE Software*, Vol. 19, No. 1, 2002, pp. 42-51.

[6] M. Weiser, "Program Slicing," *The IEEE Transactions on Software Engineering*, Vol. 10, No. 4, 1984, pp. 352-357.

[7] Y.Z. Zhang and B. W. Xu, "A Novel Formal Approach to Program Slicing," *Science in China, Series E, Information Sciences*, Vol. 38, No. 2, 2008, pp. 161-176.

[8] S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transaction on Programming Languages and Systems*, Vol. 12, No. 1, pp. 26-60.

[9] "Zeta Debugger," http://www.fyzor.com/debugger/index.htm

