Loop Detection

Peter Silberman

peter.silberman@gmail.com

Contents

Chapter 1

Foreword

Abstract: During the course of this paper the reader will gain new knowledge about previous and new research on the subject of loop detection. The topic of loop detection will be applied to the field of binary analysis and a case study will given to illustrate its uses. All of the implementations provided in this document have been written in C/C++ using *Interactive Disassembler* (IDA) plug-ins.

Thanks: The author would like to thank Pedram Amini, thief, Halvar Flake, skape, trew, Johnny Cache and everyone else at nologin who help with ideas, and kept those creative juices flowing.

Chapter 2

Introduction

The goal of this paper is to educate the reader both about why loop detection is important and how it can be used. When a security researcher thinks of insecure coding practices, things like calls to `strcpy` and `sprintf` are some of the first things to come to mind. These function calls are considered low hanging fruit. Some security researchers think of integer overflows or off-by-one copy errors as types of vulnerabilities. However, not many people consider, or think to consider, the mis-usage of loops as a security problem. With that said, loops have been around since the beginning of time (e.g. first coding languages). The need for a language to iterate over data to analyze each object or character has always been there. Still, not everyone thinks to look at a loop for security problems. What if a loop doesn't terminate correctly? Depending on the operation the loop is performing, it's possible that it could corrupt surrounding memory regions if not properly managed. If the loop frees memory that no longer exists or is not memory, a double-free bug could've been found. These are all things that could, and do, happen in a loop.

As the low hanging fruit is eliminated in software by security researchers and companies doing decent to moderate QA testing, the security researchers have to look elsewhere to find vulnerabilities in software. One area that has only been touched on briefly in the public relm, is how loops operate when translated to binaries[1]. The reader may ask: why would one want to look at loops? Well, a lot of companies implement their own custom string routines, like `strcpy` and `strcat`, which tend to be just as dangerous as the standard string routines. These functions tend to go un-analyzed because there is no quick way to say that they are copying a buffer. Due to this reason, loop detection can help the security research identify areas of interest. During the course of this article the reader will learn of the different ways to detect loops using graph analysis, how

[1]BugScan is an example of a company that has implemented "buffer iteration" detection but hasn't talked publically about it. http://www.logiclibrary.com

to implement loop detection, see a new loop detection IDA plug-in, and a case study that will tie it all together.

Chapter 3

Algorithms Used to Detect Loops

A lot of research has been done on the subject of loop detection. The research, however, was not done for the purpose of finding and exploiting vulnerabilities that exist inside of loops. Most research has been done with an interest in recognizing and optimizing loops[1].

Research on the optimization of loops has led scientists to classify various types of loops. There are two distinct categories to which any loop will belong. Either the loop will be an irreducible loop[2] or a reducible loop[3]. Given that there are two different distinct categories, it stands to reason that the two types of loops are detected in different fashions. Two popular papers on loop detection are *Interval Finding Algorithm*[1] and *Identifying Loops Using DJ Graphs*[2]. This document will cover the most widely accepted theory on loop detection.

3.1 Natural Loop Detection

One of the most well known algorithms for loop detection is demonstrated in the book *Compilers Principles, Techniques, and Tools* by Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. In this algorithm, the authors use a technique

[1]A good article about loop optimization and compiler optimization is http://www.cs.princeton.edu/courses/archive/spring03/cs320/notes/loops.pdf

[2]Irreducible loops are defined as "loops with multiple entry [points]" (http://portal.acm.org/citation.cfm?id=236114.236115)

[3]Reducible loops are defined as "loops with one entry [point]" (http://portal.acm.org/citation.cfm?id=236114.236115)

that consists of two components to find natural loops[4].

The first component of natural loop detection is to build a dominator tree out of the control flow graph (CFG). A dominator can be found when all paths to a given node have to go through another node. A control flow graph is essentially a map of code execution with directional information. The algorithm in the book calls for the finding of all the dominators in a CFG. Let's look at the actual algorithm.

Starting from the entry node, the algorithm needs to check if there is a path to the slave from the entry node. This path has to avoid the master node. If it is possible to get to the slave node without touching the master node, it can be determined that the master node does not dominate the slave node. If it is not possible to get to the slave node, it is determined that the master node does dominate the slave. To implement this routine the user would call the `is_path_to(ea_t from, ea_t to, ea_t avoid)` function included in `loop_detection.cpp`. This function will essentially check to see if there is a path from the parameter `from` that can get to the parameter `to`, and will avoid the node specified in `avoid`. Figure 3.1 illustrates this algorithm.

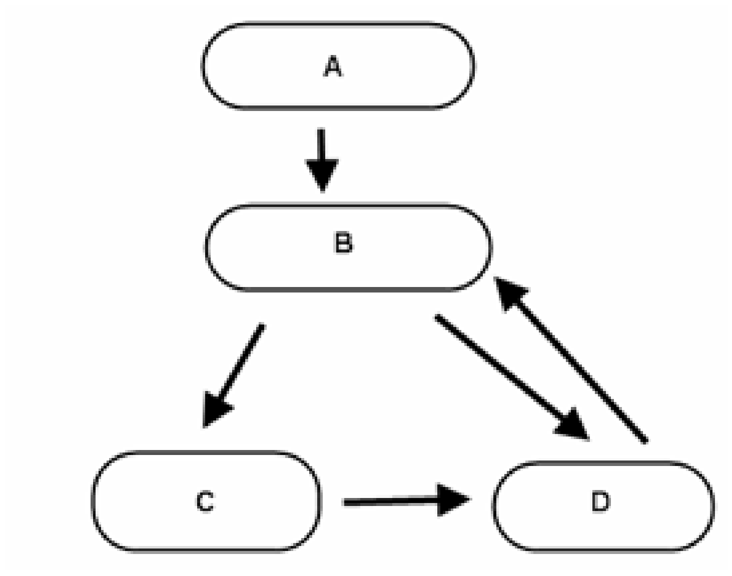


Figure 3.1: An example of a reducible loop

As the reader can see from Figure 1, there is a loop in this CFG. Let B to C to D be the path of nodes that create a loop, it will be represented as `B->C->D`. There

[4]A natural loop "Has a single entry point. The header dominates all nodes in the loop." (http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15745-s03/public/lectures/L7_handouts.pdf all loops are not natural loops

is also another loop from nodes `B->D`. Using the algorithm described above it is possible to verify which of these nodes is involved in the natural loop. The first question to ask is if the flow of the program can get from A to D while avoiding B. As the reader can see, it is impossible in this case to get to D avoiding B. As such, a call to the `is_path_to` function will tell the user that B Dominates D. This can be represented as `B Dom D`, and `B Dom C`. This is due to the fact that there is no way to reach C or D without going through B. One question that might be asked is how exactly does this demonstrate a loop? The answer is that, in fact, it doesn't. The second component of the natural loop detection checks to see if there is a link, or backedge, from D to B that would allow the flow of the program to return to node B to complete the loop. In the case of `B->D` there exists a backedge that does complete the loop.

3.2 Problems with Natural Loop Detection

There is a very big problem with natural loops. The problem is with the natural loop definition which is "a single entry point whose header dominates all the nodes in the loop". Natural loop detection does not deal with irreducible loops, as defined previously. This problem can be demonstrated in figure **??**

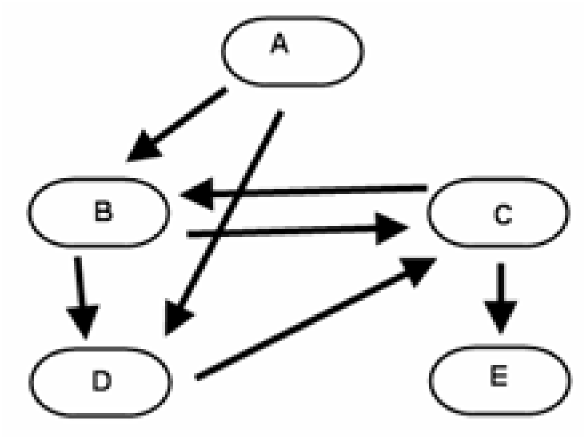


Figure 3.2: An example of an irreducible loop

As the reader can see both B and D are entry points into C. Also neither D nor B dominates C. This throws a huge wrench into the algorithm and makes it only able to pick up loops that fall under the specification of a natural loop or reducible loop[5].

[5]It is important to note that it is next that it is next to impossible to reproduce **??**F:fig2) without using nested for loops and goto's statements. For this reason it is rare that the reader will see an example of this in a binary. However, it is possible therefore the author thought it

important to mention.

Chapter 4

A Different Approach to Loop Detection

The reader has seen how to detect dominators within a CFG and how to use that as a component to find natural loops. The previous chapter described why natural loop detection was flawed when trying to detect irreducible loops. For binary auditing, the tool will need to be able to pick up all loops and then let the user deduce whether or not the loops are interesting. This chapter will introduce the loop algorithm used in the IDA plug-in to detect loops.

To come up with an algorithm that was robust enough to detect both loops in the irreducible and reducible loop categories, the author decided to modify the previous definition of a natural loop. The new definition reads "a loop can have multiple entry points and at least one link that creates a cycle." This definition avoids the use of dominators to detect loops in the CFG.

The way this alternative algorithm works is by first making a call to the `is_reference_to(ea_t to, ea_t ref)` function. The function `is_reference_to` will determine if there is a reference from the ea_t specified by `ref` to the parameter `to`. This check within the loop detection algorithm determines if there is a backedge or link that would complete a loop. The reason this check is done first is for speed. If there is no reference that would complete a loop then there is no reason to call `is_path_to`, thus preventing unnecessary calculations. However, if there is a link or backedge, a call to the overloaded function `is_path_to(ea_t from, ea_t to)` is used to determine if the nodes that are being examined can even reach each other. The `is_path_to` function simulates all possible code execution conditions by following all possible edges to determine if the flow of execution could ever reach parameter `to` when starting at parameter `from`. The function `is_path_to(ea_t from, ea_t to)` returns one (`true`) if there is indeed a path going from `from` to `to`. With both of these functions returning one, it can be deduced that these nodes are involved in the loop.

4.1 Problems with new approach

In every algorithm there can exists small problems, that make the algorithm far from optimal. This problem applies to the new approach presented above. The algorithm presented above has not been optimized for performance. The algorithm runs in a time of O(N$\hat{2}$), which carries quite a load if there are more than 600 or so nodes.

The reason that the algorithm is so time consuming is that instead of implementing a Breadth First Search (BFS), a Depth First Search (DFS) was implemented, in the `is_path_to` function which computes all possible paths to and from a given node. Depth First Search is much more expensive than Breadth First Search, and because of that the algorithm may in some rare cases suffer. If the reader is interested in how to implement a more efficient algorithm for finding the dominators, the reader should check out `Compiler Design & Implementation` by Steven S. Muchnick.

It should be noted that in future of this plug-in there will be optimizations made to the code. The optimizations will specifically deal new implementations of a Breadth First Search instead of the Depth First Search, as well as other small optimizations.

Chapter 5

Loop Detection Using IDA Plug-ins

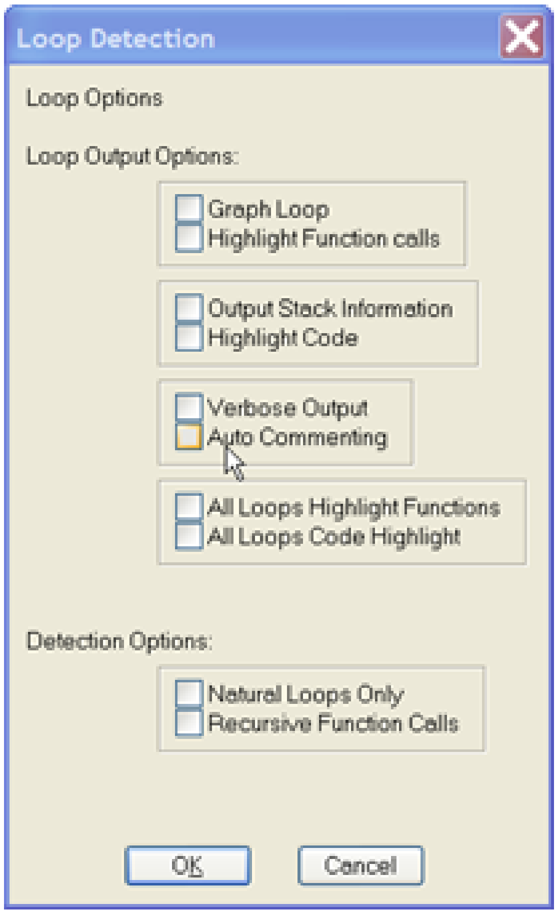
In every algorithm and theory there exists small problems. It is important to understand the algorithm presented

The plug-in described in this document uses the Function Analyzer Class (function_analyzer) that was developed by Pedram Amini (http://labs.idefense.com) as the base class. The Loop Detection (loop_detection) class uses inheritance to glean its attributes from Function Analyzer. The reason inheritance is used is primarily for ease of development. Inheritance is also used so that instead of having to re-add functions to a new version of Function Analyzer, the user only has to replace the old file. The final reason inheritance is used is for code conformity, which is accomplished by creating virtual functions. These virtual functions allow the user to override methods that are implemented in the Function Analyzer. This means that if a user understands the structure of function analyzer, they should not have a hard time understanding loop detections structure.

5.1 Plug-in Usage

To best utilize this plug-in the user needs to understand its features and capabilities. When a user runs the plug-in they will be prompted with a window that is shown in figure 5.1. Each of the options shown in figure 5.1 are described individually.

1. **Graph Loop**
   This feature will visualize the loops, marking the entry of a loop with



Loop Options

Loop Output Options:

- [ ] Graph Loop
- [ ] Highlight Function calls
- [ ] Output Stack Information
- [ ] Highlight Code
- [ ] Verbose Output
- [ ] Auto Commenting
- [ ] All Loops Highlight Functions
- [ ] All Loops Code Highlight

Detection Options:

- [ ] Natural Loops Only
- [ ] Recursive Function Calls

Figure 5.1: Loop detection plug-in options

green border, the exit of a loop with a red border and a loop node with a yellow border.

2. **Highlight Function Calls**
This option allows the user to highlight the background of any function call made within the loop. The highlighting is done within IDA View.

3. **Output Stack Information**
This is a feature that is only enabled with the graph loop option. When this option is enabled the graph will contain information about the stack of the function including the variables name, whether or not it is an argument, and the size of the variable. This option is a great feature for static auditing.

4. **Highlight Code**

This option is very similar to Highlight Function except instead of just highlighting function calls within loops it will highlight all the code that is executed within the loops. This makes it easier to read the loops in IDA View

5. **Verbose Output**
   This feature allows the user to see how the program is working and will give more information about what the plug-in is doing.

6. **Auto Commenting**
   This option adds comments to loops nodes, such as where the loop begins, where it exits, and other useful information so that the user doesn't have to continually look at the graph.

7. **All Loops Highlighting of Functions**
   This feature will find every loop within the IDA database. It will then highlight any call to any function within a loop. The highlighting is done within the IDA View making navigation of code easier.

8. **All Loops Highlighting of Code**
   This option will find every loop within the database. It will then highlight all segments of code involved in a loop. The highlighting of code will allow for easier navigation of code within the IDA View.

9. **Natural Loops**
   This detection feature allows the user to only see natural loops. It may not pick up all loops but is an educational implementation of the previously discussed algorithm.

10. **Recursive Function Calls**
    This detection option will allow the user to see where recursive function calls are located.

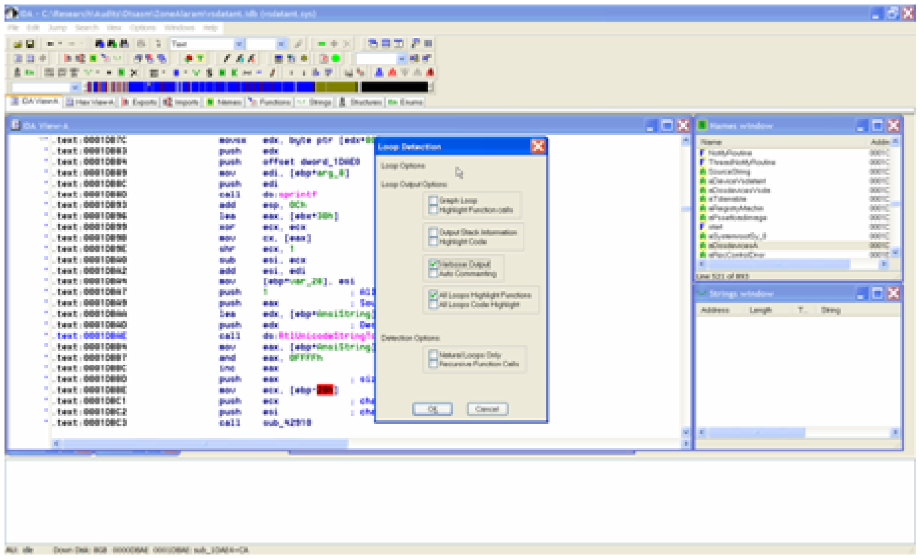
5.2 Known Issues

There a couple of known issues with this plug-in. It does not deal with `rep*` instructions, nor does it deal with `mov**` instructions that might result in copied buffers. Future versions will deal with these instructions, but since it is open-sourced the user can make changes as they see fit. Another issue is that of "no-interest". By this the author means detecting loops that aren't of interest or don't pose a security risk. These loops, for example, may be just counting loops that don't write memory. Halvar Flake describes this topic in his talk that was given at Blackhat Windows 2004[3]. Feel free to read his paper and make changes accordingly. The author will also update the plug-in with these options at a later date.

5.3 Case Study: Zone Alarm

For a case study the author chose Zone Alarm's vsdatant.sys driver. This driver does a lot of the dirty work for Zone Alarm such as packet filtering, application monitoring, and other kernel level duties. Some may wonder why it would be worthwhile to find loops in a driver. In Zone Alarm's case, the user can hope to find miscalculations in lengths where they didn't convert a signed to unsigned value properly and therefore may cause an overflow when looping. Anytime an application takes data in remotely that may be type-casted at some point, there is always a great chance for loops that overflow their bounds.

When analyzing the Zone Alarm driver the user needs to select certain options to get a better idea of what is going on with loops. First, the user should select verbose output and All Loops Highlighting of Functions to see if there are any dangerous function calls within the loop. This is illustrated in figure 5.3.



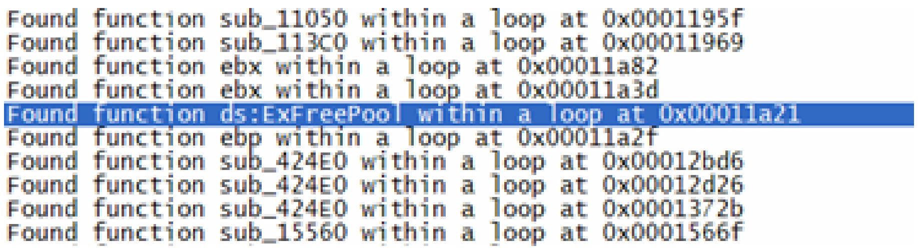
- [ ] Graph Loop
- [ ] Highlight Function calls
- [ ] Output Stack Information
- [ ] Highlight Code
- [✓] Verbose Output
- [ ] Auto Commenting
- [✓] All Loops Highlight Functions
- [ ] All Loops Code Highlight
- [ ] Natural Loops Only
- [ ] Recursive Function Calls

After running through the loop detection phase, some interesting results are found that are shown in figure 5.3.

Visiting the address `0x00011a21` in IDA shows the loop. To begin, the reader will need to find the loop's entry point, which is at:

```
.text:00011A1E                jz      short loc_11A27
```

At the loop's entry point, the reader will notice:



```
Found function sub_11050 within a loop at 0x0001195f
Found function sub_113C0 within a loop at 0x00011969
Found function ebx within a loop at 0x00011a82
Found function ebx within a loop at 0x00011a3d
Found function ds:ExFreePool within a loop at 0x00011a21
Found function ebp within a loop at 0x00011a2f
Found function sub_424E0 within a loop at 0x00012bd6
Found function sub_424E0 within a loop at 0x00012d26
Found function sub_424E0 within a loop at 0x0001372b
Found function sub_15560 within a loop at 0x0001566f
```

```
.text:00011A27                push    206B6444h ; Tag
.text:00011A2C                push    edi       ; NumberOfBytes
.text:00011A2D                push    1         ; PoolType
.text:00011A2F                call    ebp ;ExAllocatePoolWithTag
```

At this point, the reader can see that every time the loop passes through its entry point it will allocate memory. To determine if the attacker can cause a double free error, further investigation is needed.

```
.text:00011A31                mov     esi, eax
.text:00011A33                test    esi, esi
.text:00011A35                jz      short loc_11A8F
```

If the memory allocation within the loop fails, the loop terminates correctly. The next call in the loop is to `ZwQuerySystemInformation` which tries to acquire the `SystemProcessAndThreadsInformation`.

```
.text:00011A46                mov     eax, [esp+14h+var_4]
.text:00011A4A                add     edi, edi
.text:00011A4C                inc     eax
.text:00011A4D                cmp     eax, 0Fh
.text:00011A50                mov     [esp+14h+var_4], eax
.text:00011A54                jl      short loc_11A1C
```

This part of the loop is quite un-interesting. In this segment the code increments a counter in `eax` until `eax` is greater than 15. It is obvious that it is not possible to cause a double free error in this case because the user has no control over the loop condition or data within the loop. This ends the investigation into a possible double free error.

Above is a good example of how to analyze loops that may be of interest. With all binary analysis it is important to not only identify dangerous function calls but to also identify if the attacker can control data that might be manipulated or referenced within a loop.

Chapter 6

Conclusion

During the course of this paper, the reader has had a chance to learn about the different types of loops and some of the method of detecting them. The reader has also gotten an in-depth view of the new IDA plug-in released with this article. Hopefully now when the reader sees a loop, whether in code or binary, the reader can explore the loop and determine if it is a security risk or not.

Bibliography

[1] Tarjan, R. E. 1974. *Testing flow graph reducibility.* J Comput. Syst. Sci. 9, 355-365.

[2] Sreedhar, Vugranam, Guang Gao, & Yong-Fong Lee. *Identifying loops using DJ graphs.* `http://portal.acm.org/citation.cfm?id=236114.236115`

[3] Flake, Halvar. *Automated Reverse Engineering.* `http://www.blackhat.com/presentations/win-usa-04/bh-win-04-flake.pdf`