LOOKING INTO THE EYE OF THE BITS

REVERSE ENGINEERING USING MEMORY ANALYSIS

FEBRUARY, 2011

ASSAF NATIV

INTRODUCTION

During the past three years I've been developing tools for research and implementation of a new type of software analysis, which I will introduce in this paper. This new type of reverse engineering allows recovering internal implementation details using only passive memory analysis, and without requiring any disassembly. I will also discuss how to cope with the challenge that applications (including DBs) are always in a state of flux - new versions, security updates, etc., keep changing the memory structure. I will answer the question of supporting a new version of the target application without seeing it.

I will discuss the added value of this new method of internals' recovery over the more common method of disassembling and decompiling. I will also share my stockpile of common memory patterns, written in Python, and explain the vast information that can be uncovered simply by roaming around in memory land.

In my talk I will give a demonstration that will include a description of a security problem that I found in Microsoft SQL Server (published during 2009), as a result of applying this methodology. I will demonstrate how it is possible to recover the internal structures of a program as complex as a DBMS, and how one can find the important core internals that should be protected.

One major application of this technique is discussed, which is to gain the deep knowledge and understanding of the inside building blocks and design of the target application, required to implement monitoring. As far as I know this method of memory monitoring has never before been used for security purposes. This method allows us to achieve a good view of the application’s activity, on the one hand, while on the other hand minimizing the performance impact (in contrast to methods that require extensive application logging, for example). It depends on the existence of caching, pipelining and buffering of data to create a real time view of the application’s activity. When applied efficiently it can be used to protect applications from various exploits and thus can be adopted as an alternative to applying security patches to products, especially when applying the patches comes at a very high cost (e.g. extensive testing of applications, shutting down mission-critical applications, etc.).

Reverse-engineers may consider recovering internal implementations and data structures by studying memory dumps difficult or not worth the hassle. In this paper you will see that not only is this job not as complex as one may think, but it could also be more effective then traditional SRE. I will show the benefits of this work in many real world examples. I will divide this problem into four smaller subjects as following:

- Examine the tools one needs for the task
- Analyze all of the different primitives we ought to find in memory
- Discuss a simple way to define at a high level the structures and patterns to search for in memory
- Case study.

TOOLS

A lot has been said about the subject of SRE tools, and almost any debugger would be sufficient for our needs. I find the Python interactive interpreter to be the most efficient environment for carrying out research of this kind. As I research, the current status of the interpreter holds my current knowledge of the inspected target. Any piece of information can be easily accessed because it is all stored in global variables. Thanks to these benefits and many more, one can "play" with the data and try to make some sense of it. On Win32 there is the PyDBG module that enables a researcher to debug a process from a Python environment. An alternative to PyDBG would be a tool I wrote for the task called pyMint, which is freely available online.

The functionality one would be looking for in the debugger in use is:

1. Displaying memory in various ways.
2. Searching in memory in various ways.
3. Gathering as much information about the memory as possible (e.g. page attributes, memory regions, heap structures and so on).

Displaying memory dumps could be done in Binary form, Dword form, ASCII, Unicode, Graphical and more, and it's better when all modes are accessible from one integrated environment. A simple modification of the way the memory is shown can make the difference between random-looking bits and bytes and a data structure with an apparent purpose. For example here are two dumps of the same memory:

The first:

```
00  6C29 760A 6C29 760A - 0100 0000 0000 0000    l)v.l)v.........
10  0100 0000 387B C603 - 387B C603 0000 0000    ....8{..8{......
20  0000 0000 0000 0000 - 4C7B C603 4C7B C603    ........L{..L{..
30  0000 0000 0000 0000 - 0000 0000 607B C603    ............`{..
40  607B C603 0000 0000 - 0000 0000 0000 0000    `{..............
50  747B C603 747B C603 - 0000 0000 0000 0000    t{..t{..........
60  0000 0000 887B C603 - 887B C603 0000 0000    .....{...{......
70  0000 0000 0000 0000 - 9C7B C603 9C7B C603    .........{...{..
80  0000 0000 0000 0000 - 0000 0000 B07B C603    .............{..
90  B07B C603 0000 0000 - 0000 0000 0000 0000    .{..............
A0  C47B C603 C47B C603 - 0000 0000 0000 0000    .{...{..........
B0  0000 0000 D87B C603 - D87B C603 0000 0000    .....{...{......
C0  0000 0000 0000 0000 - EC7B C603 EC7B C603    .........{...{..
D0  0000 0000 0000 0000 - 0000 0000 007C C603    .............|..
E0  007C C603 0000 0000 - 0000 0000 0000 0000    .|..............
F0  147C C603 147C C603 - 0000 0000 0000 0000    .|...|..........
```

And the second:

```
 0  A76296C  A76296C        1        0              1
14  3C67B38  3C67B38        0        0              0
28  3C67B4C  3C67B4C        0        0              0
3c  3C67B60  3C67B60        0        0              0
50  3C67B74  3C67B74        0        0              0
64  3C67B88  3C67B88        0        0              0
78  3C67B9C  3C67B9C        0        0              0
8c  3C67BB0  3C67BB0        0        0              0
a0  3C67BC4  3C67BC4        0        0              0
b4  3C67BD8  3C67BD8        0        0              0
```

```
c8  3C67BEC  3C67BEC        0          0                 0
dc  3C67C00  3C67C00        0          0                 0
f0  3C67C14  3C67C14        0          0                 0
```

The first dump looks like a bunch of bytes that make no sense, while the second looks like a table in which every entry starts with two pointers followed by 3 numbers. A good (and correct, in this case) guess would be that this is an open hash table, where the first two Dwords are the next / prev pointers of the linked list and the following number is the number of items in the bucket.
Another interesting way to inspect memory is graphical, and it was used in a tool called Kartograph. This tool was created by Elie Bursztein to produce map hacks for strategy games.

MEMORY IN DETAIL

In order to classify the primitives found in memory, I've divided them into four groups.

1. Pointers
2. Data
    a. Text
    b. Time stamps
    c. etc.
3. Completely Random
4. Code

Pointers tend to have the virtue of pointing to something in memory, which helps identify them. Furthermore, the CPU handles Dword aligned addresses better, which makes the compiler, heap or OS try to make pointers aligned if possible. This means most pointers end in either 0, 4, 8 or 0xc.
"Data" is anything that is found in memory, that has a meaning such as IDs, handles, names, etc. "Data" is simply identified by prior knowledge of what it means, for instance if I know that my session ID is 0x33, finding 0x33 in a memory array would guide me in the memory maze.
Contrary to common belief, truly random numbers are hardly ever found in memory. Furthermore, even memory that is not allocated at all and is not referred to by any code is not filled with random data, but with whatever was in that memory the last time it was used. In fact, when one encounters a buffer in memory that seems to be randomly generated, it usually corresponds to encrypted data, compressed data, hash digest or a pseudo-random numbers buffer, which is helpful when trying to recover some logic.
To identify code one should be familiar with some assembly encoding. Almost every kind of CPU has it's own signatures for functions prologue / epilogue and common code. Most debuggers do a good job in separating the code from the data, and for an exotic CPU a new code searching function could be written in a matter of hours. If we take, for example, x86 and the Visual Studio compiler, we can see that almost every function ends with 0xc3 0x90 0x90 0x90 0x90 which is the RET opcode followed by four NOPs (Used for the MS detours library).

FUTURE WORK

Currently the implementation is not complete and I've focused on the aspects that were necessary for my work at Sentrigo. There is also more to considered for future work:

- Adding features such as RegExp, faster memory scanning, better memory map query and more to the Candy / Mint Python modules. Although, I didn't use any of these on my projects, other people may find these kinds of features more essential.
- Writing an Action-Script VM (Flash) in memory debugger / editor. This kind of tool could make Flash debugging and developing much more effective.
- Creating a proof-of-concept web server monitor. I do believe that the well proven security and monitoring method implemented by Sentrigo, should be used for many other applications.
- Considering malware, it is interesting to check what kind of data a virus can harvest from a target by monitoring the memory and staying completely invisible to logs and monitors. On the other hand, anti-viruses could use some of the techniques described here to search for and locate viruses and make signatures for them.

THANKS

- The Sensor team @ Sentrigo and the rest of Sentrigo for the time and effort and the great product of Hedgehog.
- Elie Bursztein for Kartograph.
- Roy Fox, Anna Trainin for proofing this paper.
- Anyone who contributes to the source.

REFS

- My Python Win32 memory inspector module: http://code.google.com/p/pymint/
- Patterns constructing and searching Python module: http://code.google.com/p/pycandy/
- My lame blog: http://nativassaf.blogspot.com/
- Python interactive interpreter that I use: http://dreampie.sourceforge.net/
- Python Win32 debugger module: http://pedram.redhive.com/PyDbg/docs/
- Kartograph: http://elie.im/talks/kartograph (Also on Defcon 2010 website)
- Microsoft detours library: http://research.microsoft.com/en-us/projects/detours/

CONTACT DETAILS

Assaf Nativ
Tel-Aviv, Israel
+972-505237809
Nativ.Assaf@gmail.com