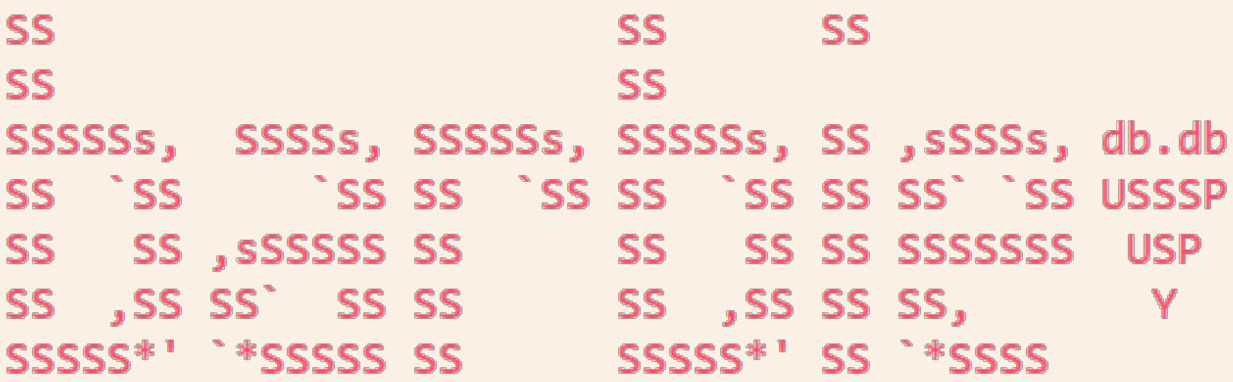


Tinkering the memory

2018-07-06 | 1.6k words | 10 minute(s) | binary analysis | reversing

50 shades of the memory

Reverse Engineering

I have been writing lots of notes on PROLOG and logical programming and people keep asking me, why I never write about reverse engineering. So I decide to give it a try.

I will start with stuff that I think are important for reverse engineering. It will be a bit different of the approaches I see on online tutorials, so I won't be mad or sad if this notes are not helpful for you. BUT I will be REALLY happy if someone finds it useful!

What am I going to write about then?

I know there are so many good sources to learn about all the basics of reverse engineering and uncountable RE101 online labs. They are all so nicely written and interesting, there is no need for me to do another one, right?

So what I am going to do is to write some notes on things I wish I knew

before, things I am learning now and things that I am curious about.

Memory Analysis

To perform memory analysis of an infected system is extremely complex and super exciting! To extract artifacts relevant to malware analysis or to perform some kind of forensic work for an incident response task is trilling me right now. I am still not sure how popular the topic is in the context of reverse engineering, but when analysing malware, a good memory analysis can help identifying the behaviour and explain how and what it was "touched" in the infected system by the malware.

As always, before starting any kind of reverse engineering tasks, you need to define the questions to be answered. In the case of analysing memory, normally you want to find out:

- Which processes were running when the snapshot was taken?
- Which processes existed before the attack / infection?
- What was the end goal of the malware?
- Any suspicious DLL modules?
- And what about the strings in the process?
- And what about URLs or IP addresses associated with a process?
- And what about the files associated with a process?
- Can we dump / extract files associated to the process?

Starting sLOWly

First of all, we should understand the memory. Lucky us, they are also built using the same logic gates you have everywhere else! There are two types of memory in our computer – one which is persistent and one that .. yeah.. In an higher level of abstraction, it is called *RAM (or random-access memory)*. So RAM is the working memory of a computer, it holds the information that is being executed by the computer like RIGHT NOW and as such it is a crucial component for a computer to operate (and a good place to act leaving not much traces).

The logic gates of the Memory

To store information, the memory uses a latch called AND-OR-LATCH, which is pretty easy to understand:

```
         +-----------------------------------+
         |       ____                        |
         +-)>- --\   `.           ____       |
                  |    >----------|   `.     |
       set--------/___.'     |`.    |     )---+---- output
       reset ---------------|  o---|___.'
                            |.'
```

Of course, inside our computers, it is a bit more complicated - i.e. we have gates for Write-Enable, we store more than one bit of data and so on. But as you see, the basic idea is simple, you just need to scale it up ;) Adding another level up to what is called *gated latch*, and grouping them in a clever way, we get what most of you already know: Registers! Our 32-bit-registers are nothing more than 32 latches built together. The same for our 64-bit-register. The difference here is that they are not side by side - as they used to be in the old 8-bit-machines - but organized as a matrix. So for the 256-bit-memory we have the 16x16 Matrix. Got it? Good!

> “Latch because the set value is persistent until a signal for setting or resetting is received.”

And how can we access the right row x column bit? We use MEMORY ADDRESSES! To convert the address into the right position, we use a component called multiplexer - which can also be built from logic gates! So to address 256-bit-memory, we use 8-bit-addresses - 4-bit for the row and 4-bit for the column. I could go on and scale up, but fast forwarding it, at some point we have 32-bit addresses, which can address a gigabyte of memory! The most important take away here is: with this address system, we are able to access ANY memory location at ANY time in ANY random order we may want / need.

Trivia: an old SRAM board had 8 modules, each with 32 submodules, which one of them with 4 smaller modules and each of them comprises a matrix 128x64 bits, making a total of 32 768 x 32 x 8 bits ~ 8 million bits

(aka 1 MB) in total!

Implementation

Lots of people don’t know how the OS takes care of the memory usage. And that’s good ;) I mean, how painful would it be if every single developer or even users needed to know it. This is one of the “magical” pieces of code hidden in the low level world. But for us, reverse engineers, it is very important to understand how it works from the inside out!

To make it easier – you can imagine the memory as a large large large array for storing data, but this is not precise, since the data is not stored linearly and all this heap/stack blub is IMPLEMENTED! But… let’s continue…

There are three different memory “types” – not 50 ;) – automatic, static and dynamic. I am going to talk about the *Stack* and the *Heap*. Simple explained one could say that in the Stack you find the local variables and in the heap you find variables that are kept alive until the end of the program, like global variables, constants and all the variables declared in C with the “static” keyword.

It would look like this in the memory:

```
                +---------+
                |         |   STACK
                +---------+
                |         |
                +---------+
                |   ...   |
                |   ...   |
                |   ...   |
                |   ...   |
                |         |
                +---------+   HEAP
                |         |
                +---------+
                |         |
                +---------+
                |         |
                +---------+
                |         |
                +---------+
```

There is more data being allocated in the memory when a process is loaded. This data can be, i.e.:

- .text – source code of the program
- .BSS (Block Started by Symbol) – not initialized a symbol
- .data – Initial data
- .heap – our global variables
- .stack – our local variables

The HEAP

I could not define it better myself, so here is a great definition from Eldad Eilam:

“A heap is a managed memory region that allows for the dynamic allocation of variable-sized blocks of memory in runtime. A program simply requests a block of a certain size and receives a pointer to the newly allocated block (assuming that enough memory is available). Heaps are managed either by software libraries that are shipped alongside programs or by the operating system.

Heaps are typically used for variable-sized objects that are used by the program or for objects that are too big to be placed on the stack. For reversers, locating heaps in memory and properly identifying heap allocation and freeing routines can be helpful, because it contributes to the overall understanding of the program's data layout. For instance, if you see a call to what you know is a heap allocation routine, you can follow the flow of the procedure's return value throughout the program and see what is done with the allocated block, and so on. Also, having accurate size information on heap-allocated objects (block size is always passed as a parameter to the heap allocation routine) is another small hint towards program comprehension."

The STACK

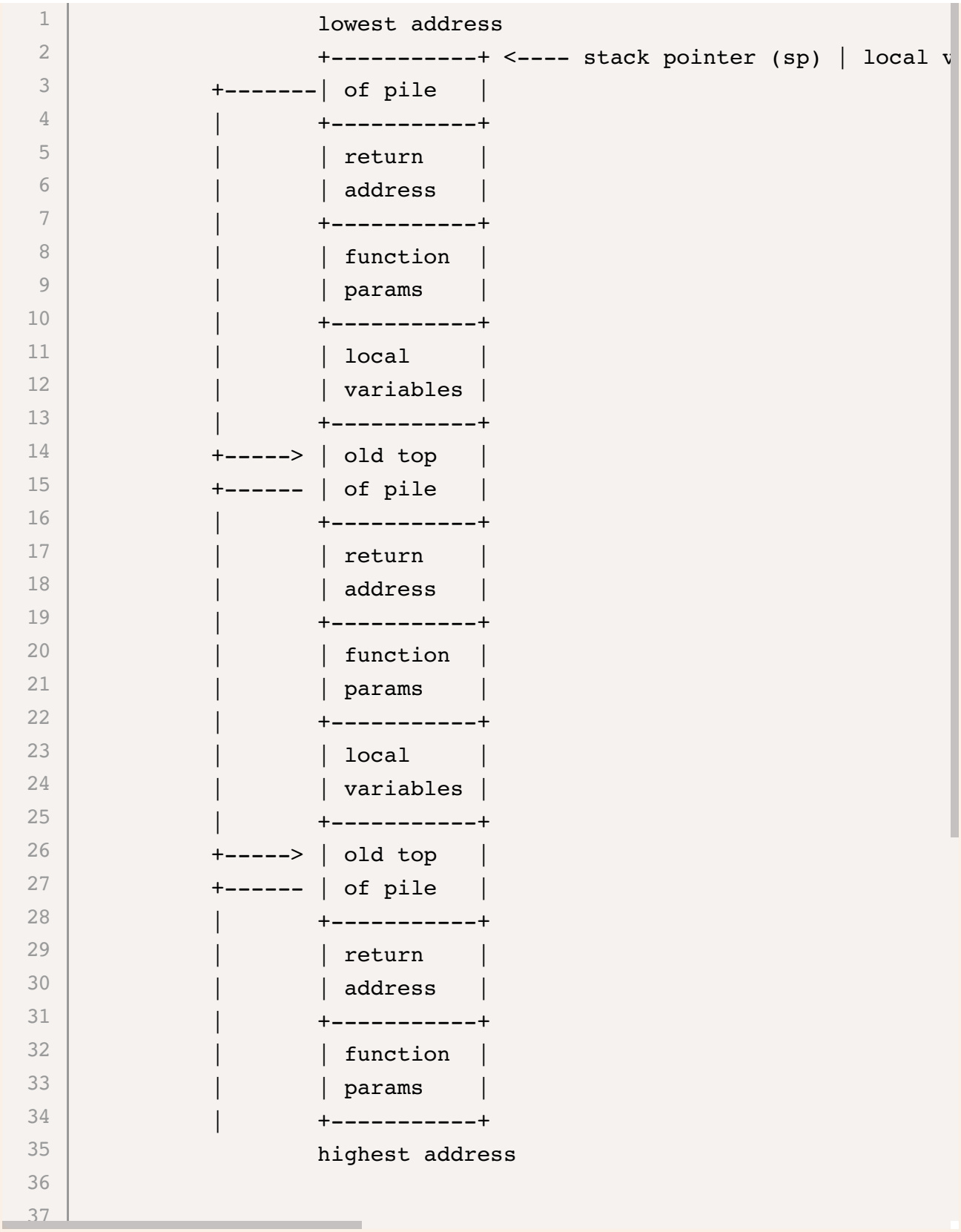
Stacks are managed as simple LIFO (last in, first out) data structures, where items are "pushed" and "popped" onto them. Memory for stacks is typically allocated from the top down, meaning that the highest addresses are allocated and used first and that the stack grows "backwards" toward the lower addresses. When a function is called during runtime, a memory block is pilled over. In this block there are references to all variables created or allocated in the called function. When the function returns, the block is deallocated.

As one can see, the variables allocated in the stack have a function lifetime.

Each pilled block is called *Stack frame*. In each frame one has the variables passed to the function parameters, the return address and a pointer called *frame pointer*, that represents the former top of the pile. Also there is a area for variables created inside the function (local variables).

```
                    +-----------+
                    | local     |
                    | variables |
                    +-----------+
                    | old top   |
                    | of pile   |  STACK
                    +-----------+  FRAME
                    | return    |
                    | address   |
                    +-----------+
                    | function  |
                    | params    |
                    +-----------+
```

This way, for each called function we pilled a stack frame above the function pile. This top of pile always contain the lowest address of the memory. This way, any address lower than the address of the top of the pile is invalid and any address higher than it is a valid stack frame.



The stack pointer points to the top of the pile. During the execution, this pointer can be updated, as we create more variables i.e. to make the movement in the stack frame, there is another pointer called frame pointer, responsible for storing a reference to the top of the pile before the function is called. Due to its position in the stack frame –

unlike the stack pointer, its position is constant – it is easy to move to the parameters until the return address and local variables.

The way the stack works is by “stacking” data into it, one “push” data in the stack, which puts the variable/data into the stack and moves the stack pointer down, you can’t remove an item from anywhere in the stack, you can always only remove (pop) the *last* item you added (pushed).

Sources

Eldad Eilam. 2005. Reversing: Secrets of Reverse Engineering. John Wiley & Sons, Inc., New York, NY, USA.

Author: barbie aka Thaís