Fast, Reliable and Runtime Protection Method Against Table Index Overflows

Piotr Bania
www.piotrbania.com

2011

Abstract

Recent security vulnerabilities show that bugs caused by insufficient or lack of boundary checks on table index variables are still very common. Successful exploitation of such bugs may lead to total system compromise just like it ended in the Microsoft SRV2.SYS SMB Negotiate ProcessID Function Table Dereference vulnerability [3, 4].

In order to protect against such attacks, we have developed a solution which decreases the probability of successful exploitation by the attacker. We are able to achieve this goal with by calculating (estimating) the table boundaries and monitoring the table index as the application executes. Our solution does not require program source code and can be implemented for both user mode and kernel mode programs. Currently the prototype works on IA-32 compatible processors.

1 Design

Our solution has only one small requirement it requires that the selected module (Portable Executable (PE) file) must have an relocation directory. This is not really any major obstacle since most of the currently deployed applications have relocation directories - all kernel mode must have relocation directories - and due to nowadays security requirements all of them should (for example in order to be compatible with Address Space Layout Randomization security mechanism (ASLR)).

Following points presents the main algorithm of our solution:

1. Detect all already loaded and any further loaded Portable Executable modules
2. Each of the find modules will be then tested for compatibility issues
3. If a compatible module is found it is being tested for the occurrence of instructions that utilize specific type of x86 addressing mode (table index)
4. If a correct instruction is found the table boundaries are being calculated (up and down limit)
5. If both of the previous steps were success the instruction is then instrumented

1.1 Instruction Detection

In order to work our solution needs to find all instructions that use specific type of x86 addressing mode (table index). Since we need to do that reliably the typical solution requires to disassemble entire Portable Executable file. However this method is slow and the code coverage is often poor. Another potential solution to this problem is to use some dynamic binary instrumentation engine like Pin [2] or DynamoRIO [1]. However they are not really useful due to similar performance issues. In order to provide a fast and reliable solution we are using our own custom method that is not so resource consuming and does not cause any noticeable performance impact.

First step of our tool is to provide entry points for the backward disassembler. To reliably conduct the process we limit our search to the executable sections of selected module which obviously should

be the typical region where the instructions reside (and can be executed). In order to get the potential entry points we iterate over the relocation directory. Since most of the instructions that refer to tables are obliged to use virtual addresses encoded as operands. In other words all of such operand locations are stored in the relocation table. From such locations we perform the backward disassembly process. It is very considerable to notice that the disassembly process can be in fact limited to the simple signature matching algorithm, which should be faster. In the current stage of our tool we are only searching for instructions like:

- `mov reg, [indexREG*4 + tableVA]`
- `call dword ptr [indexREG*4 +tableVA]`
- `jmp dword ptr [indexREG*4 +tableVA]`

However it doesn't mean that other instructions (or instructions with different scaling factor) cannot be protected. Following instructions were picked because there are typically the ones that cause potential security risk.

1.2 Calculating the table boundaries

Calculating the table boundaries or rather estimating is the second main step that ensures that our solution will work correctly without introducing false-positives alerts. At first glance it appears to be an easy task however it is not. There are to possible methods to estimate the table boundaries at this point.

First one bases on a fact that typically all of such tables should be filled with a relocable entries. In other words another set of virtual addresses. Therefore one can assume that iterating from the beginning of the table (up and down) should end when the nonrelocable entry is found. However once again this solution is far from being perfect and it causes a lot of false-positive alerts since some of the tables do not respect mentioned conditions.

Second method which is far more reliable and currently implemented in our solution is based on different approach. When new Portable Executable module is processed our engine creates a map of a regions that are requested by the relocation directory plus the entire PE is scanned for known executable code signatures (C prologue, C epilogue). When estimating the table range upward (to the lower address) we continue the scan until either a requested region is found or an executable code signature. We mark this regions as the upper table limit. However when scanning downward (to the higher address) we calculate the number of found requested regions. When the counter exceeds our const safety value we stop the scanning procedure and mark the current location as the table end. However if the executable code is found or we have reached the end of current PE section we stop the scan immediately also with marking the current location as the bottom table limit. There is one exception to this rule. When the virtual address of the table points to the region that is not initialized before the module runs (so out of the raw data) we simply mark the end of the current PE section as the end of the table. At first glance this limit appears to be very high but it often enough to terminate successful attempts of vulnerability exploitation.

1.3 Instrumentation

Instructions that have an immediate 32-bit relocable operands posses one great factor. They occupy minimum 5 bytes of memory (one minimum byte for instruction operand plus 4 bytes of the relocable offset). It is important since at this point we can freely patch it with 5 byte relative jump (trampoline) without potentially corrupting other instructions and finally crashing the application. In our case the instructions we are interested in are always 7 bytes in size. Each instruction that is considered valid and has a calculated table boundaries is then hooked. From this point the control flow is redirected to a small memory land that dynamically checks whether request exceeded the table boundaries or not. This is all done before the original instruction gets executed. If the request violates the boundary limits it is reported as an exploitation attempt and the application is terminated. Otherwise the application execution continues.

2 Drawbacks

Some instructions may use different indexing method which cannot be protected in the same manner as we have presented here. However they

are not common as the ones presented in this article.

References

[1] DynamoRIO. http://www.cag.lcs.mit.edu/dynamorio/.

[2] Pin. http://rogue.colorado.edu/pin/.

[3] CVE Database. CVE-2009-3103. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3103.

[4] Piotr Bania. SMB2: 351 Packets from the Trampoline. http://blog.metasploit.com/2009/10/smb2-351-packets-from-trampoline.html.