Mitigating Return-Oriented Programing Attacks and Other Exploitation Attempts via Secure API Execution

Piotr Bania
www.piotrbania.com

2011

Abstract

With the discovery of new exploit techniques, new protection mechanisms are needed as well. Mitigations like DEP (Data Execution Prevention) or ASLR (Address Space Layout Randomization) created a significantly more difficult environment for vulnerability exploitation. Attackers, however, have recently developed new exploitation methods which are capable of bypassing the operating system's security protection mechanisms.

Currently Return-Oriented Programming attacks are used heavily for the exploitation purposes. 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 by estimating and limiting the places from where selected (protected) API functions can be referenced. 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.

Our solution decreases the possibility of successful vulnerability exploitation without noticeable performance impact and false-positive alerts. Our work is not only limited to Return-Oriented Programming attacks. It can also harden shellcode execution and other exploitation methods as well.

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 found modules will be then tested for compatibility issues
3. If a compatible module is found it will be tested for the occurrence of protected API functions located in the module's import directory
4. Entire address space of the module is then scanned for values (memory operands) that point to the protected API address in the Import Address Table (IAT)
5. Each of such found entries is then changed, and now points to a memory (stored in random location) with newly created code land for it
6. From now on every direct reference to protected API exported by selected module is considered as attack attempt.

1.1 Protected API functions

Return-oriented programming is a known exploitation technique which allows the attacker to use

stack memory to indirectly execute previously picked instructions (so called gadgets). Our tests showed that most of the known ROP exploits always try to use `VirtualProtect` API at some point in the exploitation process. Typically the ROP exploits try to find the `VirtualProtect` address, compute the parameters, execute the `VirtualProtect` API (through the find address) and return to the "now valid" memory which contains the larger shellcode. Additionally (besides `VirtualProtect`) in our prototype we have decided to protect additional important APIs like: `VirtualProtectEx`, `VirtualAlloc`, `VirtualAllocEx`, `WriteProcessMemory`, `GetProcAddress`. However it doesn't mean other API functions cannot be protected as well.

1.2 Policy and additional informations

Our policy states that protected API functions cannot be accessed in any different way than as a result of control transfer done by the gadget we have generated. It means that even if the attacker chooses to parse the `kernel32.dll` export section manually or calculate the API distance through other API function addresses in IAT in order to get `VirtualProtect` address (for example) it will be useless since the original API will be overwritten (intercepted). Our research showed that PE modules call API functions typically by using the Import Address Table. In our tests Protected APIs functions were always accessed through the IAT entry. Obviously sometimes other API functions are called "differently" - for example the GetProcAddress API can be used in order to resolve other API function addresses without directly using the IAT entries. However such cases can still be handled. Additionally inter-modular API calls (for example in `KERNEL32.DLL` are easily detectable (always call rel - 5 bytes) and they are handled by our prototype.

1.3 Ways of API referencing

Our research showed that API functions addresses stored in IAT are typically referenced by the following instructions only:

- `mov reg, dword ptr [api_addr]`
- `call dword ptr [api_addr]`
- `jmp dword ptr [api_addr]`

Obviously the memory operand in this case always has the corresponding relocation entry. This feature allows us to predict perfectly the places where the protected API is referenced by the selected module. Since at this point we are dealing with 3 types of instructions only full disassembler is not required and a simple signature based check is enough. Depending on the instruction a specific code land is generated. Original instruction is patched with JMP REL (5 bytes) which leads to the newly generated code. Each generated code land is unique for specified instruction. Please not that the mentioned instructions are always minimum 5 bytes long - this allows us to patch them freely.

1.4 Generated code lands

Below we present newly generated code lands for specific instruction types. Each generated code block should be unique for selected original instruction.

```
push    dword ptr [esp-4]
pushfd
pushad
push    esp
push    gadget_up_border
push    gadget_down_border
push    instrVA
call    test_regs
mov     [esp+PUSHAD.reg], eax
popad
xor     reg, destVA ^ SECRET_KEY(eax)
add     esp,4
popfd
jmp next_instruction
```

Listing 1: Generated code land for mov reg, mem.

```
push    dword ptr [esp-4]
pushfd
pushad
push    esp
push    gadget_up_border
push    gadget_down_border
push    instrVA
call    test_regs
mov     [esp+PUSHAD.reg], eax
popad
xor     eax, destVA ^ key
add     esp,8
push    nextINSTR_VA (if this was a call only)
```

```
jmp eax

with RET instrumentation:
call    eax     ; dest
original_instructions_without_ret
pushfd
pushad
push    [esp+PUSHAD_SIZE+PUSHFD_SIZE]
call    verify_ret
popad
popfd
original_ret
```

Listing 2: Generated code land for call/jmp/callrel.

Notes:

- `test_regs` function - if registers are correct (their values do not overlap with calculated borders, `[esp-4]` value is also checked here) function returns key in EAX otherwise it never returns
- `verify_ret` function is responsible for checking (filtering) the return address

1.4.1 RET instrumentation

RET instrumentation is additional feature that is used when following sequence of code is found:

```
call dword ptr [api]
org instructions
pop / leave (must occur)
ret (in 15 bytes between call and orinal instrs)
```

Listing 3: Sequence suitable for additional RET check.

The main idea here is to check the return address before the control transfer will occur. Tests include checking whether the return address is a part of loaded module or whether there is a call opcode before it.

1.5 Checks and Borders

Since we have disabled the possibility of direct API usage the attacker may still try to use a gadget (part of original code) that executes protected api through our generated code land. Such attacks can be (in most of the cases) filtered by following checks:

- checking `[esp-4]` value
- testing if any of the general purposes regs are inside the calculated borders

The method of calculating the borders relies on the information gathered from the relocation section (see `CalculateGadgetBorderUp` / `CalculateGadgetBorderDown` for details).

2 Weaknesses and Drawbacks

One of the possibilities for the attacker would be to use gadgets that meet the border requirements (such places are hard to find) or to use native API functions (typically API parameters for such functions are harder to compute, plus the syscall numbers change between various Microsoft Windows systems). Main drawback of this solution is that it may not work with modules that do not have relocation directory, are packed or are using custom way of executing (referencing) protected API functions.