Security Mitigations for Return-Oriented Programming Attacks

Piotr Bania
www.piotrbania.com

2010

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.

In this paper we present a short summary of novel and known mitigation techniques against return-oriented programming (ROP) attacks. The techniques described in this article are related mostly to x86-32[1] processors and Microsoft Windows operating systems.

1 Introduction

In order to increase the security level of the operating system, Microsoft has implemented several mitigation mechanisms, such as DEP and ASLR. Data Execution Prevention (DEP) is a security feature that prohibits the application from executing code from non-executable memory area. To exploit a vulnerability, an attacker must find a executable memory region and be able to fill it with necessary data (e.g., shellcode instructions). Generally, achieving this goal using old exploitation techniques is made significantly more difficult with the addition of the DEP mechanism. As a result, attackers improved upon the classic "return-into-libc" technique and started using return-oriented programming (ROP) [3, 7] to bypass Data Execution Prevention.

Techniques like ROP are still based on the attacker understanding memory layout characteristics, leading Microsoft to implement Address Space Layout Randomization (ASLR) as a countermeasure. ASLR renders the layout of an application's address space less predictable because it relocates the base addresses of executable modules and other memory mappings. In order to bypass DEP protection mechanism ROP technique was introduced. In this article we present novel and known mechanisms which are created specifically to prevent attackers from exploiting vulnerabilities based on the ROP method. Presented mitigations will be divided in two general categories:

- Compiler-level mitigations — mitigations that can be only applied by the compiler or linker.
- Binary-level mitigations — mitigations that can be applied without knowing the source code of the protected code fragment.

2 Return-oriented Programming

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). Typically each gadget ends with the x86 subroutine return instruction[2] (`RET`), which further transfers the execution to the next gadget or the payload itself. For more information regarding the return-oriented programming technique please refer to [1, 3, 7].

[1] Some of the techniques can be also applied on other architectures, albeit some of them are only available for x86-32 family (e.g., the ones based on creating new segment descriptors).

[2] However other instructions may be used as well like `jmp reg`, `call reg` etc.

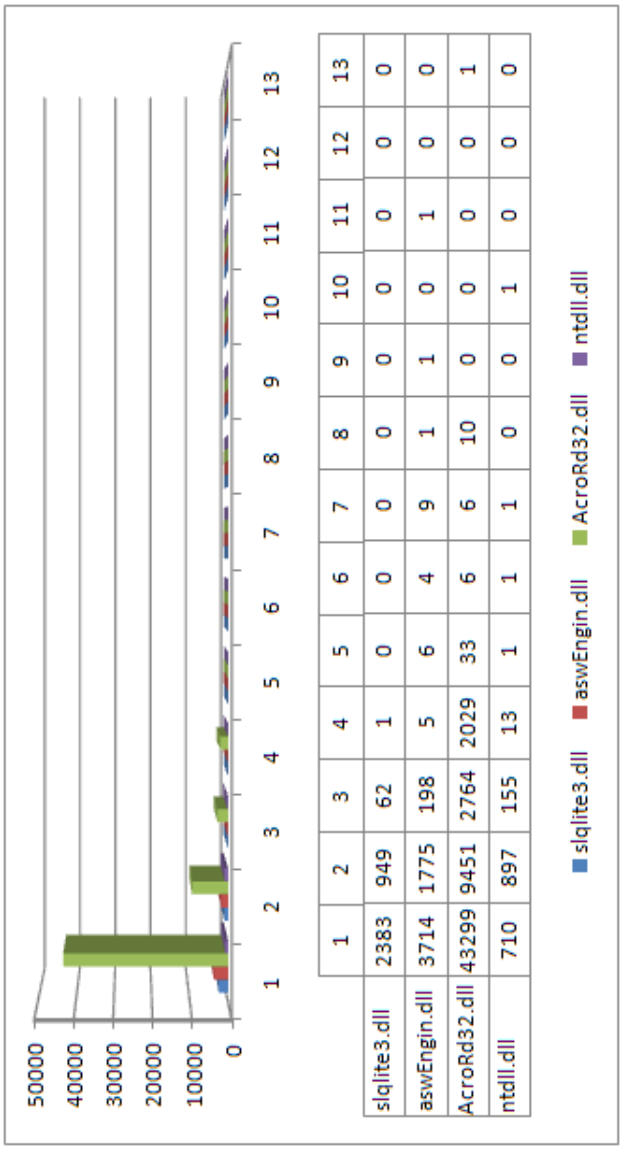
3 Compiler-level mitigations

In this section we present ROP protection mechanisms which can be applied at the compiler-level. However this doesn't mean they are not implementable at the binary-level - they are simply substantially easier to implement at the compiler-level. We will also try to underline advantages and disadvantages of described mechanisms.

The biggest disadvantage of compiler-level mitigations is the fact that they require code recompilation in order to be effective. It is often hard to quickly implement such kind of changes in the real world.

3.1 Call-Ret relations

As previously stated, most gadgets use return instructions to transfer execution control to another gadget or payload. In order to find useful gadgets, attackers scan the process memory or the binary module for return instruction opcodes and, after such opcode is found, they try to perform backward disassembly in order to decide whether following gadget is useful (correct) or not. Return instruction opcodes can often be found in the middle of different instructions. Results, however, show that most of the time original return instructions `RET` are used. Typically they also represent the highest number of return opcodes found in the entire module's executable area (cf. Figure 1). For the remainder of this article `RET` instructions emitted in the original program's code will be named as "original return instruction".



	1	2	3	4	5	6	7	8	9	10	11	12	13
slqlite3.dll	2383	949	62	1	0	0	0	0	0	0	0	0	0
aswEngin.dll	3714	1775	198	5	6	4	9	1	1	0	1	0	0
AcroRd32.dll	43299	9451	2764	2029	33	6	6	10	0	0	0	0	1
ntdll.dll	710	897	155	13	1	1	1	0	0	1	0	0	0

Figure 1: `RET` opcode offsets in sample modules (offset equal to 1 indicates that this is an original `RET` instruction).

3.1.1 Testing for CALLs

In typical applications, every procedure (function) is executed by using call-procedure instruction. Every `CALL` instruction saves procedure linking information on the stack and branches to the procedure specified by the destination operand. Our ROP mitigation technique relies on a fact that each return address popped from the stack by the `RET` instruction is preceded by `CALL` instruction. When a ROP attack occurs the return address points to another gadget (or finally a payload). It is unlikely that an attacker will be able to pick the return addresses preceded by `CALL` instruction operands (see

Table 1 for details). Testing for CALL instructions located before the return address popped from the stack should be a reliable method against ROP attacks.

Module Name	N_1 [#]	N_2 [#]
ntdll.dll	6528	138 (2.11%)
ieframe.dll	45232	2109 (4.66%)
bib.dll	5966	317 (5.31%)
aswEngin.dll	50895	1547 (3.03%)

Table 1: Number of gadgets preceded by relative, memory indirect, register indirect procedure-call instruction ("minimal/not extended" addressing mode assumed).

Where:

- N_1 represents total number of gadgets
- N_2 represents number of gadgets preceded by the procedure-call instruction
- gadget represents a valid single instruction or sequence of instructions without any special filtering applied regarding the gadget usefulness

However, the method itself has some drawbacks. CALL instructions can be encoded in various ways (relative, absolute, indirect), which can influence the scanner's performance and also the potential reliability of this method. Secondly, only original return instructions can be protected. In other words, using different instructions (like indirect jumps or calls) for linking gadgets will be not detected. On the other hand, the CALL opcode checking method can be based on opcode-frequency statistics, which could decrease the potential performance slowdown. Additionally, since only specified (valuable for attacker) return instructions can be protected, this should have a positive influence for the program's performance.

3.1.2 Emitting magic values

This method was introduced by the Pax Team [5] and it relies on emitting magic bytes after every CALL instruction and testing them at the function epilogue, as shown in Listing 1.

This method seems to be more reliable than the method described in Section 3.1.1, although it also has some major drawbacks. First of all, the TEST instruction isn't neutral for the application context's state, since the EFLAGS register is modified by this instruction. This flaw[3], however, can be easily fixed by simply emitting JMP OVER_MAGIC instruction after each CALL. A more serious limitation of this method is the fact that every module used by the application would have to be created (compiled and linked) with the same MAGIC value. This is necessary since execution transfers may occur from one module to another[4].

```
callee
epilogue:
        mov register,[esp]
        cmp [register+1],MAGIC
        jnz .1
        retn
.1: jmp esp

caller:
        call callee
        test eax,MAGIC
```

Listing 1: Protection of the execution flow changes via the return instructions.

Since this approach would be almost impossible to implement in the real world there is another solution which can be used here. We propose that Windows' Portable Executable loader be responsible for synchronizing every MAGIC value after each system boot (and after specified module is loaded). This would of course require creating a new section (or some new, specific data directory) with all the MAGIC values offsets that should be updated by the executable loader.

3.2 Obfuscating instructions

This approach addresses the problem where the RET instruction opcode is a part of different instruction (typically it is located among the first 1-3 bytes, not including instruction opcode). Owing to our tests and external sources [7] most of such opcodes are found in the ModR/M byte. A second large source of RET opcodes is found in immediate dis-

[3]Whether this is a flaw or not depends mostly on the application binary interface; in most cases the caller is responsible for saving the flags.

[4]Modules that don't perform execution transfers to other modules can be left "unsynchronized".

placements. In order to prevent from effectively using such cases in the ROP attack we propose that every instruction with `RET` opcode inside of its body will be obfuscated in a special manner. Of course control transfer instructions or any other instructions that use immediate data offsets are an exception to this rule since the immediate displacements are calculated by the linker. The potential obfuscation can be done in following fashion:

- If `RET` opcode is found in the first byte after the original instruction opcode, a jump land should be emitted just before this instruction. Such jump land should consist of a short unconditional jump instruction and a land (up to 16 bytes) of `INT3` or other worthless for attacker single byte instructions. Such emitted instructions will never be executed by the original program flow because of the unconditional jump, which transfers the execution directly to the potentially dangerous instruction. Such action should decrease the number of effective gadgets used for creating the ROP chain.
- If `RET` opcode is spotted in immediate constant values such instruction should be obfuscated for example by splitting `ADD REG,IMM32` into two `ADD` instructions where the `IMM32` operand for both of them would be free of return instruction opcodes. Of course special care must be taken regarding the `EFLAGS` register state after each such transition.
- If `RET` opcode is found in ModR/M byte, which indicates using `EAX` register as destination operand and `EBX` register as source operand (e.g., `MOV EAX,EBX`), such instructions can be transformed into equivalent form which doesn't include return instruction opcodes. For example `MOV EAX,EBX` ↔ `PUSH EBX; POP EAX` (`0x53 0x58`).

As previously mentioned, the presented solutions can only be applied to instructions that do not use immediate displacements, as those are handled by the linker.

4 Binary-level mitigations

In this section we present mitigations against ROP attacks that can be applied without any information of the program's original source code. All mitigations included in this section can be implemented at the binary-level.

```
1: mov esp,eax
   ret

2: xchg eax,esp
   ret

3: add esp,<number>
   ret
```

Listing 2: Typical stack pivot sequences.

4.1 Stack Encapsulation

To make a ROP attack work, the attacker must be able to point the stack pointer into the controlled data. In typical stack-buffer overflow vulnerabilities this is not needed, but in other vulnerabilities (e.g., heap-overflow) this is often a must. In order to achieve this goal, the attackers use the so called stack pivot sequence [1]. Listing 2 shows some commonly used stack pivot sequences. Our mechanism tries to take advantage of this information.

When a new thread is created, operating systems reserve some necessary space for its stack memory. Stack borders are described in the `INITIAL_TEB` structure which is passed in one of parameter of `NtCreateThread` function. Additionally stack borders are also available in the Thread Information Block (`FS:[0x04]` - top stack, `FS:[0x08]` - current bottom stack). When the attacker uses the pivot sequence he typically exceeds the stack border limits set by the thread initialization procedure. The methods described in the following sections were designed to recognize this behavior. Similar support must be taken when dealing with fibers, since they also use separate stacks.

4.1.1 New stack segment descriptor

Microsoft Windows systems allow usermode applications to create their own local descriptor table (LDT). Most current operating systems use the flat memory model, where there is no need to create additional segments for every running application. This would be in fact a step back to the old segmented memory model. On Windows platforms, in usermode, all segments' base addresses are equal to

```
xor eax,eax
lea edi,[esp+VALUE]
stosd
stosd
...
```

Listing 3: Typical program instructions.

zero, except the one pointed by the FS register (the GS segment register is not used[5]). In our mitigation mechanism we have developed two approaches that protect the system against the stack pivoting technique. Our initial technique was to create a stack segment descriptor each time new a thread is created with a base address equal to the stack bottom and limit corresponding to stack size. After the new segment is created we initialize the SS segment register with a new value.

This method however has a big drawback, which is explained on the listing below (Listing 3).

The LEA instruction is responsible for initializing the EDI register with the effective address of ESP+VALUE. However, the value that will be stored in the EDI register is still relative to the stack segment base address (which is not null in our case). The problems start with instructions that don't use the SS segment register for addressing purposes. For example, the STOSD instruction uses the ES segment register; its execution will end with an access violation, since the base address of the segment pointed by ES segment register is different. In other words the LEA instruction does not honor the segment registers when calculating the effective address.

To resolve this issue we were forced to change the base address of the newly created stack segments. To avoid unnecessary access violations, the stack segment base address was set to zero and its limit was set to the stack's top value. This has some disadvantages, since the attacker would need to initialize new stack pointer value with an address higher than the segment limit to trigger the mechanism. Most of the time, however, newly allocated buffers have higher addresses since the thread's stack memory allocation was done earlier (there are a few obvious exceptions to this rule). Each time the attacker tries to exceed the boundaries of the current stack segment a general protection fault occurs and, at this point, our filtering procedure decides if the selected process is being exploited and needs to be terminated.

As a side note, there is one small problem with this method. Instructions that use the EBP register for memory addressing are also using the stack segment specified by the current segment selector. This means that if the EBP is not related to the stack memory and the destination address exceeds the stack segment boundaries a general protection fault will occur. Such cases however can be easily filtered and the execution can be resumed after emulating the faulting instruction.

Countermeasures In order to bypass the stack encapsulation protection, an attacker would need to initialize the stack pointer with a lower memory address than the stack's top value. For example attacker can heap-spray the memory and then cause the application to create a new thread that will be used to trigger the vulnerability. By doing this attacker fake stack will be below the stack base. Another way would be to execute a gadget that reinitializes the stack segment with the original value (constant between Windows versions) by, for example, executing a POP SS instruction. To disable this attack we are constantly monitoring the value of the SS segment register, and we reinitialize it every time execution returns from a system call (since kernel reinitializes the segment registers values before the control is returned to the usermode).

4.1.2 Monitoring stack pointer changes

Another approach for detecting the stack pivoting technique is to monitor the stack pointer value at crucial areas. For example, instead of setting another segment for stack space we can hook important offensive API functions (e.g., VirtualAlloc, VirtualProtect) and test the stack pointer value there. Obviously, there is no guarantee that the attacker wouldn't be able to restore the original stack pointer before using such API functions. To improve the security level of this protection mechanism we also propose that newly allocated memory regions (or memory regions with changed page protection rights) with executable pages should be marked as non-executable[6]. Now the page marked

[5] This is true for x86-32 architectures only

[6] this requires having a CPU with NX bit support

as non-executable by our mechanism will work as a decoy. If the processor is trying to execute the non-executable page (page protection was previously changed by our mechanism) then we firstly apply our filtering procedure which tests the stack pointer value. If everything is correct, the executable rights are re-enabled and the execution is continued — the entire mechanism works like a one-time decoy.

4.2 Code Encapsulation

The ROP technique, just like any other, has some strategic points. One of those is the fact that the attacker must know the virtual addresses of the used gadgets. Because so, the ASLR mechanism successfully obstructs the exploitation process. However, in some cases the attacked application is either not compatible with ASLR or just uses external modules which do not support the ASLR mechanism. There are also cases where the attacker is able to leak or guess the wanted virtual address, rendering the ASLR mechanism relatively easy to bypass. In this section we present a mechanism which will take advantage of this (ROP) technique's weak spot.

As stated in Section 4.1.1, Windows systems allow usermode applications to create their own local descriptor tables. In this mechanism we propose that each loaded module's code in the application's address space (including the main module) will have a separate segment for code sections[7], as Figure 2 shows. Each time a execution transfer between modules or execution transfer using a full virtual address (including module imagebase value) occurs, a general protection fault will happen. At this point the filtering procedure decides whether this execution transfer attempt is valid or an attack attempt.

This method has some drawbacks:

- A lot of control transfers are done through API calls and since they require a code segment switch a general protection fault is thrown every time such action occurs. Since this has a negative impact on the application's performance, entire import address table entries should be redirected to specific API stubs as shown in Listing 4.

[7] this mechanism is a bit similar to PaX SEGMEXEC [6]

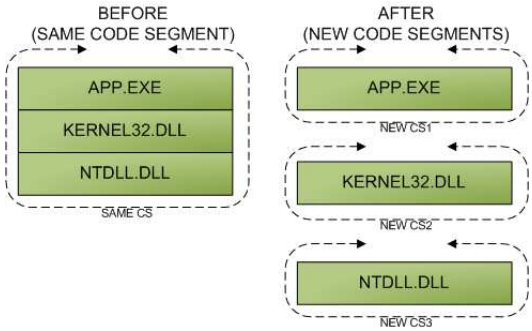


Figure 2: Not-encapsulated and encapsulated modules inside of the process memory.

This solution should successfully decrease the negative performance impact because of the decreased number of GP faults. However, this is only one aspect of the problem, since the requested API must be able to return correctly to the specified location which is outside the current code segment. There are a few ways to solve this issue. One of the potential solutions can be based on faking the return address in the API land stub and then recalculating the correct address when `RET` instructions cause the GP fault. Every potential solution here, however, will decrease the program performance. Additionally, since API lands can be only generated either before the base address of the specified module or attached to the end of it the code segment borders need to be expanded as well (at least in cases that don't overwrite module's memory).

- Special care must be taken when dealing with case-switch offsets since they also contain virtual addresses that don't apply to the new code segment limits. This issue can be partially resolved with using module relocation information and applying some heuristic scanning mechanism. All case-switch offsets found should be recalculated again and now point to relative addresses. However, some modules do not provide relocation information which makes dealing with such cases hard and probably slow.
- Some Portable Executable modules like `SHELL32.DLL` are pretty large (8MB - 16MB). This causes some additional problems, since if

```
CALL DWORD PTR DS:[0x406010]

original memory at 0x406010:
(ptr to user32.CreateWindowExA)
00406010 A9 E4 37 7E

patched memory at 0x406010:
00406010 dd offset api_land1

api_land1:
jmp user32_cs:rel_offset
```

Listing 4: Example implementation of IAT redirection.

a function callback address located in different module has a virtual address somewhere between this 0-X MB range and some instruction will try to execute this virtual address the mechanism will fault. This is caused because the function's callback virtual address is located within the limits of the current code segment, and therefore the GP fault does not occur. This is a major drawback since it will likely lead to an application crash. Potential workarounds for this issue would be to disallow (or reserve) the memory located at the 0-X MB range. However, this would require an interaction with the system's Portable Executable loader.

- As explained before, every time kernel returns control to the usermode code segment registers are reinitialized with default values. Thus, the protection mechanism needs to re-initialize them as well each time such action happens.
- Additionally the number of segment descriptors is limited however this is not a problem for most of the applications since the number of loaded modules is not high.
- Special care must be taken when dealing with original code hooks, since such cases exists in some of the applications (for example in `IEXPLORE.EXE` this is done by the `IEFRAME.DLL` module).

Countermeasures The attacker would have to restore the original `CS` register value by, for example, returning into a `RETF` instruction. To protect against such attacks, firstly the current `CS` segment will be monitored at the crucial program places and secondly all the newly generated code segment selector values will be pseudo-randomized.

4.3 Code Decoys

This approach requires a processor with NX bit [2] support. The method itself is rather simple and it can be described in few steps:

1. Mechanism setups a page fault filter and also module filter, which activates each time after a new module is mapped into process memory.
2. All code sections from selected module found in the process memory are relocated to random memory address with the preservation of the section alignment (see Figure 3).
3. After the relocation is done original code sections are marked as not-executable (see Figure 3).
4. Each time a page fault occurs because of an execution attempt of not-executable memory, the filtering procedure decides if it should recalculate the instruction pointer and continue the execution or to kill the process because of exploitation attempt.

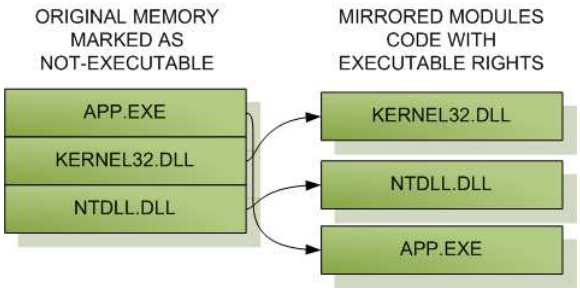


Figure 3: Code decoys created from original modules.

A similar idea was also used by the PaX Team in RANDEXEC mechanism and also by Matt Miller in the WehnTrust project [4].

To improve the performance of this mechanism, special care must be taken when dealing with case-switch offset tables — this was already mentioned

in Section 4.2. Additional performance improvements can be achieved with import address table redirecting, not unlike the idea explained in Section 4.2. It is important, however, to point out that this idea can also lower the protection level of this mechanism.

Countermeasures The attacker would have to guess or leak the mirrored code address.

5 Acknowledgments

Author would like to thank Brad Spengler, Matt Miller and the Kryptos Logic team for helping with this article.

6 Conclusion

In this article, a number of promising techniques which can be used against the return-oriented programming attacks were presented.

Most of the implementation problems of such mitigations are directly linked to a heavy performance impact. This is also a major factor in discouraging incorporation of these (and other) ROP mitigations into the selected platforms. Our security mitigations do not solve the problem of using return-oriented programming attacks completely, but they can effectively trammel and limit their usage.

References

[1] Dino A. Dai Zovi. Practical Return-Oriented Programming. `http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf`.

[2] Eric Grevstad. CPU-Based Security: The NX Bit. `http://hardware.earthweb.com/chips/article.php/3358421`.

[3] Erik Buchanan, Ryan Roemer, Stefan Savage, Hovav Shacham. Return-oriented Programming: Exploitation without Code Injection. `https://www.blackhat.com/presentations/bh-usa-08/Shacham/BH_US_08_Shacham_Return_Oriented_Programming.pdf`.

[4] Matt Miller. WehnTrust a Host-based Intrusion Prevention System (HIPS) for Windows 2000, XP, and Server 2003. `http://wehntrust.codeplex.com/`.

[5] PaX Team. Pax Future. `http://pax.grsecurity.net/docs/pax-future.txt`.

[6] PaX Team. Segmexec. `http://pax.grsecurity.net/docs/segmexec.txt`.

[7] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.